Dyninst on the RISC-V: Binary Instrumentation in Support of Performance, Debugging, and Other Tools

Cheng-Hsun Angus He
cahe@wisc.edu

Ronak Chauhan
ronak@cs.wisc.edu

James A. Kupsch
kupsch@cs.wisc.edu

Hsuan-Heng Wu
hwu337@wisc.edu

Barton P. Miller
bart@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison
Madison, WI 53706

***Abstract*— Binary instrumentation provides the ability to instrument and modify a program after the compilation process has completed. Operating on the binary level allows instrumentation of the program as it was produced by the compiler. In addition, it can operate on programs or libraries for which you may not have the source code. Binary instrumentation is the foundation for a wide variety of tools, including those for performance profiling, debugging, tracing, architectural simulation, and digital forensics. Dyninst is a free and open-source suite of toolkits for building binary analysis and instrumentation tools for architectures that include the x86, ARM, and Power. It is used in tools produced by industry, academia and research labs. This paper describes our efforts to port Dyninst to the RISC-V architecture. We discuss the challenges presented by the RISC-V, our approaches to solving them, and the status of Dyninst on the RISC-V.**

Keywords—binary instrumentation, binary rewriting, dynamic instrumentation

I. Introduction to Binary Instrumentation

Binary instrumentation is a powerful technique that allows you to directly manipulate binary code by inserting, deleting or modifying instructions in the binary code. The advantage of binary instrumentation is that it operates directly on the binary. Operating directly on the binary does not require access to or even have access to the source and works on the executable as was generated by the compiler. Binary instrumentation has a long proven history in areas such as performance profiling [1], taint analysis [28], debugging [5], architectural simulation [9] and malware detection [15][19][24]. The usefulness of this approach has continued with the development of binary instrumentation and profiling tools for GPUs [32][34].

For example, if you wanted to trace every function entry and exit, or every memory access, or even every stack memory reference, you can easily create a modified version of your executable file that contains such instrumentation.

There are two types of binary instrumentation based on when the instrumentation happens: (1) static instrumentation, called *binary rewriting*, meaning that the binary code is modified and a new executable or library file is created, and (2) *dynamic instrumentation,* where the modification of the binary happens while the program is running. Static and dynamic instrumentation are illustrated in Fig. 1. Binary instrumentation was launched in the early 1990's, with ATOM [14] as the earliest tool to implement binary rewriting and Dyninst [8][17][22] as the earliest tool to support dynamic instrumentation.

Over the years, there have been many tools that have been developed to do binary instrumentation, including DynamoRIO [7], EEL [20], Pin [21], Valgrind [23], angr.io [31], GTPin [32], and NVBit [34].

Binary instrumentation tools work in a variety of ways:

Coding patching: Tools that use code patching directly instrument the existing binary. They do this by creating a new version of the block or whole function that contains the instrumentation and *relocating* this code, i.e., placing this instrumented code in a patch area called a *trampoline*. The original code is then overwritten to contain a branch to the instrumented version of the code, and the instrumented code is terminated by a branch back to the original code. The advantage of such tools is that they keep much of the original code intact and only change what will be instrumented. The disadvantage is that these tools incur some extra control flow transfers to and from the trampolines. These tools also need (and often benefit from) strong semantic analysis of the code to understand its structure. Dyninst is an example of such a tool.

Code caching: Tools that use code caching relocate every basic block before it is executed. If the block will have instrumentation, then the relocated version is modified before it is copied to the code cache. All code is copied and executed from the cache. The advantage of such tools is that they are structurally simpler to build. However, they have the overhead of relocating each block of code and do not benefit from code analysis. Valgrind and PIN are examples of such tools.

Hoisting: Tools that use hoisting convert the binary to a common intermediate representation (IR) such as LLVM IR, modify the IR, and then regenerate the code to form a new binary. The advantage of such tools is that they do not require code generation or patching functionality. All that is done by the tools that supports the IR. However, such tools must completely understand every instruction in the binary so that it can be hoisted to the IR (which can often be complex or even

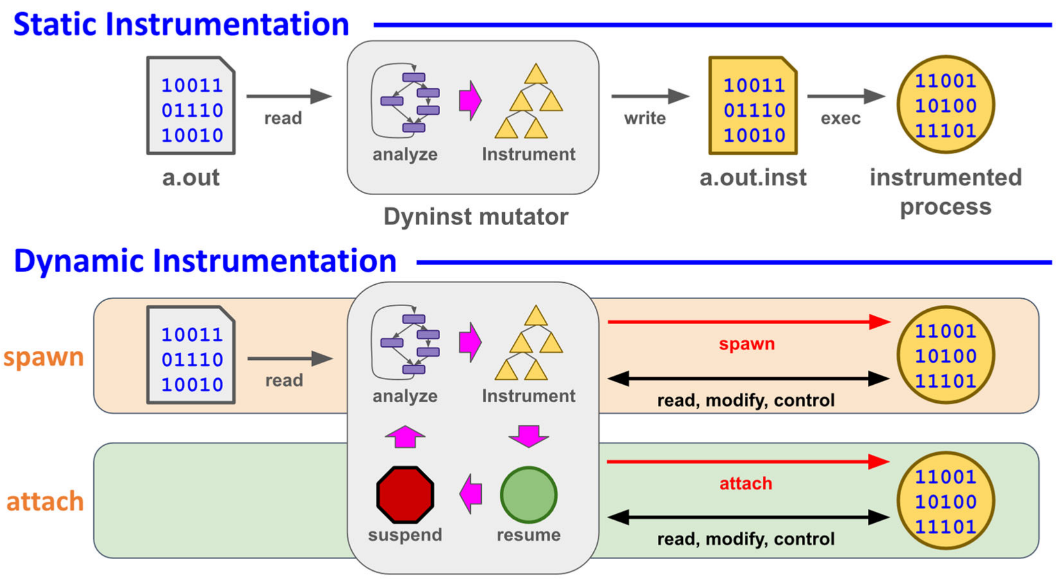

Fig. 1. The variants of binary instrumentation. Static binary instrumentation reads a binary, performs analysis and instrumentation and then creates a new binary. Dynamic binary instrumentation has two forms that differ in when it occurs. In the first form the binary is analyzed and instrumented and the resulting process is spawned. In the second form an already running process is attached to. In both case further process control, analysis and code modification may occur.

impossible to do). Tools that use hoisting include llvm-mctoll [35] and BinRec [3].

II. Dyninst

Dyninst is a binary analysis and instrumentation toolkit that provides a machine independent interface to machine level binary (executable or library) analysis, instrumentation (code modification), and platform independent process control. The abstract interface allows Dyninst-based tools to operate without any specific knowledge of the structure of the ISA of the processor.

The analysis and instrumentation capabilities of Dyninst are used by tools such as Rice University's HPCToolkit [1], University of Oregon and ParaTools TAU [30], Barcelona Supercomputing Center's Paraver [25], Lawrence Livermore National Labs Stack Trace Analysis Debugging Tool (STAT) [5], AMD OmniTrace [2], and Red Hat SystemTap [18]. Note that Dyninst is **free and open source**, based on an LGPL license and hosted on GitHub [13].

Dyninst is unique among tools in that it does both analysis and instrumentation. Dyninst uses a deep semantic analysis of the code to allow a more complex understanding of the code to be instrumented and to enable more efficient instrumentation. It first performs control- and data-flow analysis on the binary (the *mutatee*) to create a control flow graph (CFG) and dataflow graph (DFG) of the mutatee; information in these graphs is used during the instrumentation process. Dyninst's code modification is based on safe transformations of the program's CFG so that instrumented binary will have valid control flow transitions [6]. The dataflow information also plays important role in the instrumentation process, including *liveness analysis* that finds free registers, called *dead* registers, that can be used to create efficient instrumentation that can avoid the need to save current program values. Dyninst's dataflow analysis is also used to analyze pointer-based control flow (such as is used in jump tables, virtual function tables, and function pointers) to generate a more complete (and thus, more accurate) CFG. This unique combination of binary analysis and instrumentation allows more robust and sophisticated instrumentation to be crafted.

Currently Dyninst supports analysis and instrumentation of the x86, ARM64, Power, and (in progress) AMD GPU architectures. In the past Dyninst has supported many other architectures such as Sun SPARC, DEC Alpha, HP PA-RISC, MIPS, and Intel Itanium.

Dyninst is broken down into toolkits to allow the functionality to be used separately or collectively. Some components are specific to a particular ISA (such as the InstructionAPI) or operating system (such as the

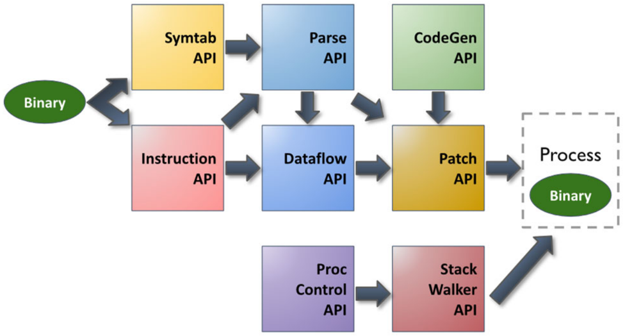

Fig. 2. The components of Dyninst and the use relationships between the components. The direction of the arrows indicates the flow of information.

ProcControlAPI), while others operate on platform independent abstractions, such as the ParseAPI and DataflowAPI.

Dyninst analyzes the binary opportunistically in that it can operate on a binary without symbols or debugging information (a stripped binary) but will use information that is optionally available in a binary, such as debugging symbols, to increase the effectiveness and accuracy of its code analysis.

Instrumentation is performed based on code *snippets* and instrumentation *points*. A snippet is an abstract representation of the code to be inserted into the binary. This code is specified by a machine independent abstract syntax tree (AST) [8]. The AST types include operations for reading or writing memory, registers or variables; performing basic logical and arithmetic operations; calling functions; and branching, including conditional branching.

A point is a location in the program where instrumentation will be inserted. Points include

- Low level abstractions such as individual instructions.
- Function level abstractions such as call site, function entry, and function exit.
- Control flow graph abstractions such as branch-taken and branch-not-take edges, loop back edges.

Code snippet insertion is a basic Dyninst operation that takes a tuple (**P**, *AST*), where **P** is a vector of instrumentation points where the instrumentation will be inserted and *AST* is the root of the tree that represents the code to be inserted. Dyninst will convert the AST to native code, optimize the code when possible, generate new versions of the blocks or functions that have been modified, and patch a branch into the original code to jump to the modified code.

Dyninst is organized as a library of toolkits. The toolkits and their relationships are shown in Fig. 2. The rest of this section will briefly describe the key functionality of each component.

A. *Code Analysis Toolkits*

The first four components perform analysis of the binary.

SymtabAPI provides an abstract representation of how a binary program is structured and stored in a file. So Dyninst can provide a platform independent interface to formats such as ELF, DWARF, PE, and PDB. It provides access to symbol table information, sections that contain the binary code and data, relocation information, and debugging data.

InstructionAPI provides an abstract representation of machine code instructions. It parses the instruction into an ISA independent representation that includes the opcode; operands including whether it is read or written and an AST that represents the addressing calculation; abstract instruction types such as call, branch, return, arithmetic; set of registers read and written by the instruction.

ParseAPI creates and provides access to the CFG. It initiates parsing of the machine code in the binary using a fast parallel algorithm to create the annotated CFG that include functions, loops, jump tables, and basic block structure of the binary. The parallel algorithm has allowed Dyninst to efficiently parse binaries that have more than a gigabyte of machine code. It uses a traversal algorithm [29][33] to construct basic blocks and determine function and loop boundaries. Parsing starts from known entry points – such as the program entry point and function entry points from symbol tables – and follows the control flow transfers to build the CFG and identify more entry points. Not all code will necessarily be found by traversal parsing due to unresolvable pointers used in control flow instructions. Thus, parsing may leave gaps [16] in the binary where code may be present but has not yet been identified. Dyninst attempts to resolve these gaps using advanced dataflow analysis techniques such as slicing and jump table analysis, and speculative parsing based on machine learning [27].

DataflowAPI annotates the CFG with dataflow information, effectively creating a DFG. The dataflow information has two general uses. First, Dyninst uses this information to increase the accuracy of the control flow analysis. Second, these operations are available to users of the DataflowAPI to build more advanced tools and custom analyses. The supported analyses include register liveness, stack height analysis, forward slicing (instructions affected by data), backward slicing (instructions that affected data), and loop analysis. Dataflow analysis requires semantic information about what each instruction calculates, currently sourced from ROSE [26], SAIL [4], and hand-crafted semantic descriptions.

B. *Instrumentation Toolkits*

The next two components perform instrumentation. Since snippets and points are architecture independent abstractions, most tools that use these interfaces are architecture independent.

While it is also possible to specify instrumentation as raw machine code in an array of bytes, this mechanism is rarely needed in Dyninst so its use is discouraged.

CodeGenAPI transforms the machine independent AST representation to architecture-specific instruction sequences.

PatchAPI does snippet insertion. It is responsible for modifying the code so that space is allocated for the instrumented code and that control is transferred to the instrumented code and back.

The final two components are used only for dynamic instrumentation, performing operations on processes (i.e., running programs).

ProcControlAPI is an operating system independent interface to process control, providing debugger-like functionality that is able to attach to a running process or start a process; read and write the memory of the process; suspend or resume a process and its threads; insert breakpoints; catch user events like signals; and detect process and thread creation.

StackwalkerAPI allows users to collect a call stack (known as walking the call stack) and access information about each invoked function's stack frames including return addresses. Each stack frame is a record of an executing function (or function-like object such as a signal handler or system call). Stack walking can be quite tricky on code generated by modern compilers, as stack frames can appear in a variety of forms or even missing altogether due to code optimizations.

III. Porting Dyninst to RISC-V

RISC-V is an open standard ISA known for its simplicity and extensibility. While RISC-V is widely used in embedded systems and microcontrollers, it has also gained popularity in high performance computing. Making Dyninst available on the in RISC-V provides a wide variety of functionality, including a pathway to port many types of tools, as mentioned in Section II. We have ported Dyninst to RISC-V, allowing for binary analysis and instrumentation on this architecture.

In this section, we start by discussing some characteristics of RISC-V that affect the porting of Dyninst. We then describe the RISC-V implementation of each Dyninst component, discussing implementation details, issues we encountered, and solutions we developed. Last, we outline the current status of the RISC-V port as well as several directions for future work.

A. Characteristics of RISC-V that affect porting

The main difficulty in porting Dyninst to RISC-V lies in the very reason that makes RISC-V appealing for hardware design: extensibility and simplicity. While RISC-V's extensibility allows flexible hardware implementation, it provides challenges for tools that have to generate and process the machine code. While hardware designers can choose to implement only extensions that are relevant to their hardware, Dyninst must support a broad range of extensions to handle the wide variety of real-world binaries.

In addition, RISC-V is a RISC architecture. While RISC architectures simplifies the implementation of hardware, they can increase the complexity of analyzing the code as some high-level functionality requires more instructions than would be needed on a CISC architecture. The simple instructions on a RISC architecture are also likely to cause some instructions top be used for multiple purposes. While this challenge also exists in other RISC architectures such as ARM or Power (which are also supported by Dyninst), RISC-V's base instruction set is significantly smaller than that of ARM or PowerPC, causing these instruction sequences to appear more frequently.

1) Extension-based ISA

One of the most distinctive features of RISC-V is its modular design. Unlike traditional ISAs that define a fixed, monolithic instruction set, RISC-V defines minimal base ISAs and several optional extensions. A base ISA defines the minimum set of integer instructions required to implement a fully functional RISC-V processor, so any RISC-V implementation must implement a base ISA. Besides the base ISA, RISC-V offers a wide range of extensions that allow hardware designers to implement only the extensions necessary for their hardware.

The extensibility of RISC-V, however, means that Dyninst needs to be aware of the extensions supported by the mutatee. If the mutatee does not support a specific extension, Dyninst should not generate instrumentation code using any instructions from that specific extension.

Additionally, new extensions are introduced and ratified every year [12]. To keep up with the fast paced change, Dyninst's RISC-V port needs to be written with extensibility in mind. Components that are ISA-dependent, including instruction parsing and code generation, need to be modular so that adding a RISC-V extension into Dyninst does not require manually changing multiple parts of the source code.

2) The C Extension (Compressed Instructions)

The C Extension is a widely used extension that offers 2-byte versions of several commonly used 4-byte standard instructions. The goal of compressed instructions is to reduce code size and improve memory usage and efficiency.

Despite the benefit of reduced code size, compressed instructions sometimes create space issues for binary instrumentation. For example, Dyninst needs to insert jump instructions to redirect control flow to instrumented code. However, Dyninst sometimes cannot use the compressed jump instruction `c.j` for this purpose because its target offset range is limited to $[-2^{12}, 2^{12})$ bytes. If the target offset exceeds this limit, Dyninst needs to fall back to a standard 4-byte jump instruction. In exceptional cases, such as functions that are shorter than four bytes, these longer jumps cannot be used. Dyninst will try to choose the most efficient jump sequence in each case, ultimately resorting to the inefficient 2-byte trap instructions in the worst case (which, fortunately, does not occur often).

3) Multi-use Control Flow Instructions

Another challenge we have faced is the multiple uses of control flow instructions in RISC-V. RISC-V defines only two instructions for unconditional branches, `jal` and `jalr`, different from instruction sets like x86 that have different instructions for unconditional jumps, function calls, and function returns. As a result, a single RISC-V branch instruction serves multiple purposes. For example, the `jalr` (jump and link register) instruction is used for unconditional jumps, function calls, function returns, and jump tables. Therefore, Dyninst needs to detect the context in which the jump instruction is being used to correctly determine its higher level purpose.

B. Toolkit-by-toolkit discussion

In this section, we provide a detailed walkthrough of the RISC-V implementation component by component, focusing on how Dyninst addresses RISC-V's wide variety of extensions and instructions.

1) SymtabAPI

SymtabAPI is responsible for parsing symbol tables and object file headers of ELF (Executable and Linkable Format) binaries. The RISC-V ABI specifies some definitions that are unique to RISC-V ELF that require special handling.

The first important field is `e_flags`, which is used to describe processor-specific properties of an ELF binary. This includes

- `EF_RISCV_RVC`: Defines whether compressed instructions are present
- `EF_RISCV_FLOAT_ABI_SINGLE`: Defines whether single-precision floating points are present
- `EF_RISCV_FLOAT_ABI_DOUBLE`: Defines whether double-precision floating points are present

The original usage of `e_flags` is to allow the linker to prevent linking ELF files with incompatible ABIs. From Dyninst's point of view, `e_flags` provide information about whether the binary is compiled for a processor that supports the compressed instruction, single-precision floating point, and double-precision floating point extensions. Thus, this field is extracted by SymtabAPI to determine whether these extensions are supported.

In addition, the ABI defines a custom section named `.riscv.attributes`. This section contains compatibility information that a linker or runtime loader needs to correctly execute RISC-V binaries, such as the target architecture string, which contains information about what extensions the binary supports. SymtabAPI will parse the `.riscv.attributes` section and obtain the value of the target architecture string to determine all the extensions that the binary uses.

While `.riscv.attributes` can be found in most binaries compiled by GCC or LLVM, it is not a mandatory section. If an ELF binary lacks this section, SymtabAPI obtains extension information from `e_flags`, as `e_flags` is present in all ELF files.

2) InstructionAPI

We base our RISC-V instruction parsing on the Capstone library [10]. Capstone is widely used and supports architectures such as x86, ARM, PowerPC, and RISC-V. There are several reasons why Capstone is ideal for instruction parsing. First, Capstone is fast and efficient: it can parse a large amount of assembly code efficiently due to its optimized disassembly engine. Second, Capstone provides detailed information about instruction operands, including whether an operand is a register, immediate value, or memory, whether an operand is read or written, whether the operand is implicit, and the memory access size for memory operands. Third, Capstone is actively maintained and updated, so when new instructions are introduced, Capstone will be promptly updated.

The version of Capstone required by InstructionAPI is v6.0.0-Alpha or above. Prior to this version, Capstone lacked support for operand read and write information. We extended Capstone's RISC-V capabilities to address this problem, and our pull request was accepted and merged into Capstone as part of the v6.0.0-Alpha release.

The set of extensions supported by a processor is called a *profile*. Currently, Capstone supports the RV64GC profile, one of the most commonly used profiles for general-purpose computing. RV64GC stands for 64-bit RISC-V architecture with support for the G (Generic) and C (compressed instruction) extensions, where the G extension is a set of base and standard extensions necessary for general-purpose computing, including the I (integer), M (integer multiplication and division), A (atomic), F (single-precision floating point), D (double-precision floating point), Zicsr (control and status register instructions), and Zifencei (instruction-fetch fence) extensions.

Capstone is planning to support new extensions such as vector instructions, which will be required by the RVA23 profile, the future ISA that most processors will support.

3) ParseAPI

ParseAPI is responsible for constructing CFGs with basic blocks, loops, and functions. While most parts of ParseAPI are platform agnostic, it still needs to recognize specific instruction sequences from different architectures to construct correct CFGs. For example, ParseAPI needs to identify function prologues and epilogues to correctly define function boundaries. Similarly, ParseAPI needs to correctly identify branch instructions to recognize basic blocks and control flow.

For RISC-V, the most challenging part is recognizing what high-level operation is represented by the `jal` and `jalr` instructions. RISC-V uses these two instructions for the following purposes:

- Function returns: Function return in RISC-V is equivalent to an unconditional jump to the return address stored in a link register, the register that contains the return address. The link register is `x1` by convention, though other registers may be (and are) used.
- Function calls: Like unconditional jumps, if the relative offset fits within the range supported by `jal`, compilers generate `jal` for function calls. Otherwise, compilers load the jump target to a register and generate a `jalr`.
- Unconditional jumps: If the relative offset fits within the range supported by `jal`, compilers generate `jal` for unconditional jumps with the link register of `x0`, a special register whose value is always zero. Otherwise, compilers load the jump target to a register and generate a `jalr`, again with the link register of `x0`.
- Tail calls [11]: A tail call is a function call-return optimization that uses a jump instruction instead of a call instruction to avoid stack frame setup and tear-down when a call instruction is the last operation in a function. In this case, a simple jump actually represents a function call.
- Jump tables: Compilers typically implement jump tables using `jalr`, where the target address is computed at runtime based on an index and loaded into a register.

Thus, given a `jal` or `jalr` instruction without any context, ParseAPI cannot determine what types high-level operation it represents only by the instruction opcode.

In addition, the valid target offset range of `jal` is limited. When the target offset exceeds `jal`'s limit, compilers will generate multi-instruction code sequences instead. For instance, compilers might generate an instruction sequence using the `auipc` (add upper immediate to PC) instruction that first loads the upper 20 bits of the jump target from the current PC to the register, followed by a `jalr` instruction that handles the lower 12 bits of the jump target:

```
# Assign t0 to PC + the upper 20 bits of offset
auipc t0, offset1
# Jump to t0 + the lower 12 bits of offset
jalr x0, offset2(t0)
```

If we only focus on the `jalr` instruction without considering the previous `auipc`, this instruction appears to be an indirect jump to `t0 + offset2`. However, `t0` can in fact be determined because it is loaded by the preceding `auipc` instruction. ParseAPI needs to examine the whole instruction sequence to correctly identify it as an unconditional jump. Note that the above sequence is only one of the possible sequences that the compiler might generate for multi-instruction jumps or calls. Different compilers may generate these sequences in different ways, which makes reliably recognizing these instruction sequences challenging.

Due to the above reasons, ParseAPI analyzes the link register and the target address to correctly identify what kind of branch the current `jal` or `jalr` represents. Obtaining the link register of `jal` and `jalr`, and the target address of `jal` can simply be done by examining its instruction operands. However, obtaining the target address of `jalr` is more challenging: ParseAPI tries to determine the exact value of the target register by performing a backward slice on it. If the result of the slicing yields a constant, ParseAPI will first check whether the constant (i.e., the target address) lies in a valid code region. If so, ParseAPI checks:

- If the target address lies within the same function, and the link register of the current instruction is `x0`, ParseAPI identifies it as an unconditional jump.
- If the target address points to other functions, and the link register of the current instruction is `x0`, ParseAPI identifies it as a tail call.
- If the target address points to the entry point of a function, and the link register of the current instruction is not `x0`, ParseAPI identifies it as a function call.
- If the target address is in a valid code region, and the previous instruction is a function call, and the link register of the function call is the same as the target register, ParseAPI identifies it as a function return.
- If none of the above cases are valid, ParseAPI performs jump table analysis [22] on the current `jalr` instruction. If it succeeds, ParseAPI identifies it as a jump table.
- If the jump table analysis fails, ParseAPI treats the `jalr` as unresolvable, meaning that the jump or call target cannot be determined symbolically.

4) DataflowAPI

The DataflowAPI provides common types of dataflow analysis such as register liveness, slicing, and loop analysis. While most parts of the DataflowAPI are platform agnostic, slicing requires instruction semantics that are architecture dependent. For x86, ARM, and PowerPC, instruction semantics are obtained from C++ classes derived from the ROSE project. While ROSE provides instruction semantic support for several architectures, it lacks support for RISC-V.

To support RISC-V, we derived the instruction semantics from the official formal specification of the RISC-V architecture, which is written in SAIL. SAIL is a language that provides a high-level executable model of instruction set architectures. Its design involves rigorous formal analysis, so it is suitable for generating emulators and theorem-prover definitions. From the perspective of binary analysis tools, this kind of formal semantic information is precisely what is needed for dataflow analysis that relies on rigorous instruction semantics to track how values propagate through the code.

However, SAIL presented some practical challenges for Dyninst. First, the SAIL language is designed to be easily parsed and executed within the OCaml ecosystem, but it limits interoperability with tools developed in other languages. In addition, the SAIL language is designed for formal verification, so the formal RISC-V SAIL definition contains many details related to error handling, such as memory alignment checks and jump target validation logic. These checks are important for formal verification or emulators, but not for dataflow analysis.

To address these issues, we developed a pipeline that acts as a source-to-source compiler from SAIL to the C++ instruction semantic classes used in DataflowAPI. The first stage of this pipeline is an OCaml script that parses the SAIL semantics and generates a simplified JSON representation of the instruction semantics. This JSON format serves as an intermediate representation that contains essential semantics of each instruction without extraneous error-handling code. The second stage of the pipeline is a script that reads the simplified JSON representation and generates C++ instruction semantic classes.

The main advantage of this pipeline design is that if new RISC-V extensions are proposed and later added to RISC-V SAIL, we only need to rerun the whole pipeline again to generate the updated C++ instruction semantic classes.

5) CodeGenAPI

CodeGenAPI is responsible for generating instrumentation code, making use of the extension information obtained from the SymtabAPI to make sure that only compatible instructions are generated.

RISC-V lacks basic instructions for some common operations, such as loading an immediate value into a register. For example, to load a 64-bit immediate value into a register, we need to generate the `lui` instruction first to load a value into the upper 20 bits of the register. Then, a sequence of `addi` (add immediate) and `slli` (shift logic left immediate) is generated to construct the immediate value.

In addition, RISC-V instructions often handle immediate values in ways that are not straightforward, such as being shifted or encoded. As a result, these nuances make generating immediate value handling one of the more error-prone aspects of code generation.

6) ProcControlAPI

The purpose of ProcControlAPI is to provide an OS independent interfaces to common process control operations, typical based on the debugging system call interface. On Linux, ProcControlAPI is implemented using the `ptrace` system call and `/proc` file system.

We have begun working on ProcControlAPI and encountered the issue that the `ptrace` system call implementation is relatively primitive in RISC-V compared to other architectures. For example, the single-stepping functionality is not implemented for RISC-V, meaning that ProcControlAPI needs to emulate single-stepping on the software level: single-stepping must be emulated by a series of breakpoints created by ProcControlAPI, which decreases performance.

7) StackwalkerAPI

While we have not started porting StackwalkerAPI to RISC-V, we anticipate several challenges due to how RISC-V handles the stack frame register. Although the RISC-V ABI designates register `x8` as the frame pointer register, many compilers choose to use `x8` as a general purpose register. That is, most compilers handle stack frames using only the stack pointer register. The StackwalkerAPI has a plugin-based architecture so that it can support multiple types of frame structures. The instruction set and compilers for RISC-V will require new "frame steppers" to be designed for the RISC-V platform.

C. *Current status*

Dyninst fully supports binary analysis for the RV64GC profile. While static instrumentation is a work in progress, it is largely functional.

For RISC-V binary analysis, Dyninst can parse the `.riscv.attributes` section, parse and analyze instructions, analyze control flow and construct CFGs, and perform jump table analysis, forward slicing, backward slicing, and loop analysis.

For RISC-V instrumentation, many features of the CodeGenAPI are fully functional. For example, Dyninst can create variables, arithmetic operation snippets, memory operation snippets, function snippets, and insert instrumentation code at the entry or exit points of functions, branches, and loops.

D. *Future work*

Our immediate future work will focus on completing the CodeGenAPI and do more testing to make it robust. Our first release is planned for 4Q2025 with the static (binary rewriting) instrumentation features. In 1Q2026, we will complete porting ProcControlAPI and StackwalkerAPI to complete support for dynamic binary instrumentation.

In the future, we will extend Dyninst to support the RVA23 profile, which is a new profile on which new RISC-V systems are standardizing as a minimal set of features. This profile includes many new extensions, such as the vector extension and integer conditional extension. Supporting new extensions should be straightforward once Capstone adds support for it. We can generate the C++ semantic classes that we need to interpret new instruction semantics using the SAIL data the same way we did for other extensions.

IV. Conclusion

In this paper, we presented our work on porting Dyninst to RISC-V. We addressed the challenges posed by RISC-V's simple and extensible design, including its modular ISA design, compressed instructions, and multi-use control flow instructions. We described changes made to each Dyninst component to support parsing, analysis, and code generation for RISC-V binaries. Our RISC-V port adopts a modular design, making it easier to add support for new RISC-V extensions.

For the RV64GC profile, the binary analysis features of Dyninst are complete. The static binary instrumentation is feature complete, and it is undergoing testing. The dynamic binary instrumentation is a work in progress as we need to complete the process toolkits, including ProcControlAPI and StackwalkerAPI. Once they are complete, Dyninst will provide full binary analysis and both static and dynamic instrumentation for the growing RISC-V ecosystem.

References

[1] L. Adhianto, S. Banerjee, M. Fagan, M. Krentel, G. Marin, J. Mellor-Crummey and N. R. Talent, "HPCToolkit: Tools for Performance Analysis of Optimized Parallel Programs", *Concurrency and Computation: Practice and Experience* **22***,* 6, April 2010.

[2] Advanced Micro Devices, Inc, "Omnitrace: Application Profiling, Tracing, and Analysis*",* Retrieved August 10, 2025, from https://rocm.docs.amd.com/projects/omnitrace/en/latest/doxygen/html/index.html

[3] A. Altinay, J. Nash, T. Kroes, P. Rajasekaran, D. Zhou, A. Dabrowski, D. Gens, Y. Na, S. Volckaert, C. Giuffrida, H. Bos and M. Franz, "BinRec: Dynamic Binary Lifting and Recompilation", *15th European Conference on Computer Systems (EuroSys)*, online, April 2020.

[4] A. Armstrong, T. Bauereiss, B. Campbell, A. Reid, K. E. Gray, R. M. Norton, P. Mundkur, M. Wassell, J. French, C. Pulte, S. Flur, I. Stark, N. Krishnaswami and P. Sewell, "ISA Semantics for ARMv8-A, RISC-V, and CHERI-MIPS", *46th ACM SIGPLAN Symposium on Principles of Programming Languages (POPL)*, Cascais, Portugal, January 2019.

[5] D.C. Arnold, D.H. Ahn, B.R. de Supinski, G.L. Lee, B.P. Miller and M. Schulz, "Stack Trace Analysis for Large Scale Debugging", *21st IEEE International Parallel and Distributed Processing Symposium (IPDPS),* Long Beach, California, March 2007.

[6] A. R. Bernat and B. P. Miller, "Structured Binary Editing with a CFG Transformation Algebra", *19th Working Conference on Reverse Engineering*, Kingston, ON, Canada, October 2012.

[7] D. L. Brening, "Efficient, Transparent, and Comprehensive Runtime Code Manipulation", *Ph.D. Dissertation,* Massachusetts Institute of Technology, Cambridge, MA, Order Number AAI0807735, September 2004.

[8] B. Buck and J. K. Hollingsworth, "An API for Runtime Code Patching", *International Journal of High Performance Computing Applications* **14**, 4, November 2000.

[9] H.W. Cain, K.M. Lepak, and M.H. Lipasti, "A Dynamic Binary Translation Approach to Architectural Simulation, *Workshop on Binary Translation (WBT),* Philadelphia, Pennsylvania, October 2000.

[10] Capstone Project, "Capstone The Ultimate Disassembler", Retrieved August 2025, from https://www.capstone-engine.org/

[11] W. D. Clinger, "Proper Tail Recursion and Space Efficiency", *ACM Conference on Programming Language Design and Implementation (PLDI),* Montreal, Canada, June 1998.

[12] E. Cui, T. Li, and Q. Wei, "RISC-V Instruction Set Architecture Extensions: A Survey," *IEEE Access*, February 2023.

[13] Dyninst Project, "Dyninst source code GitHub repository", https://github.com/dyninst/dyninst

[14] A. Eustace and A. Srivastava, "ATOM: a Flexible Interface for Building High performance Program Analysis Tools", *USENIX 1995 Technical Conference*, New Orleans, Louisiana, January 1995.

[15] W. Fang, B.P. Miller and J.A. Kupsch, "Automated Tracing and Visualization of Software Security Structure and Properties", *9th International Symposium on Visualization for Cyber Security (VizSec),* Seattle, Washington, October 2012.

[16] L.C. Harris and B.P. Miller, "Practical Analysis of Stripped Binary Code", *Workshop on Binary Instrumentation and Applications (WBIA)*, St. Louis, Missouri, September 2005.

[17] J. K. Hollingsworth, B. P. Miller, J. Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools", *Scalable High-Performance Computing Conference (SHPCC)*, Knoxville, Tennessee, May 1994.

[18] B. Jacob, P. Larson, B. H. Leitao and S. A. M. M. da Silva, "SystemTap: Instrumenting the Linux Kernel for Analyzing Performance and Functional Problems", *IBM Redbooks*, January 2009.

[19] E. R. Jacobson, A. R. Bernat, W. R. Williams and B. P. Miller, "Detecting Code Reuse Attacks with a Model of Conformant Program

Execution", *International Symposium on Engineering Secure Software and Systems (ESSoS),* Munich, Germany, February 2014.

[20] J. R. Larus and E. Schnarr, "EEL: Machine-Independent Executable Editing", *17th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, La Jolla, California, June 1995.

[21] 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", *26th, ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI)*, Chicago, Illinois, June 2005.

[22] X. Meng and B.P. Miller, "Binary Code is Not Easy", *25th International Symposium on Software Testing and Analysis (ISSTA)*, Saarbrucken, Germany, July 2016.

[23] N. Nethercote and J. Seward, "Valgrind: a Framework for Heavyweight Dynamic Binary Instrumentation", *28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI),* San Diego, California, June 2007.

[24] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua and A. D. Keromytis, "Retrofitting Security in COTS Software with Binary Rewriting", *26th IFIP TC-11 International Information Security Conference (IFIP SEC),* Hamburg, Germany, June 2011**.**

[25] V. Pillet, J. Labarta, T. Cortes and S. Girona, "PARAVER: A tool to Visualize and Analyze Parallel Code", *18th World Occam and Transputer User Group Conference (WoTUG-18)*, Manchester, England, 1995.

[26] D. Quinlan, "Rose Compiler Support for Object-Oriented Frameworks", *Parallel Processing Letters*, **10**, 2-3, September 2000.

[27] N. E. Rosenblum, X. Zhu, B. P. Miller and K. Hunt, "Learning to Analyze Binary Computer Code", *23rd AAAI Conference on Artificial Intelligence (AAAI)*, Chicago, Illinois, July 2008.

[28] E. J. Schwartz, T. Avgerinos and D. Brumley, "All You Ever Wanted to Know about Dynamic Taint Analysis and Forward Symbolic Execution (but Might Have Been Afraid to Ask)", *32nd IEEE Symposium on Security and Privacy*, Oakland, California, May 2010.

[29] B. Schwarz, S. Debray, and G. Andrews, "Disassembly of Executable Code revisited", *9th IEEE Working Conference on Reverse Engineering (WCRE)*, Richmond, Virginia, October 2002.

[30] S. S. Shende and A. D. Malony, "The Tau Parallel Performance System", *International Journal of High Performance Computing Application*s **20**, 2, May 2006.

[31] 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", *37th IEEE Symposium on Security and Privacy*. San Jose, California, May 2016.

[32] A. Skaletsky, K. Levit-Gurevich, M. Berezalsky, Y. Kuznetcova and H. Yakov, "Flexible Binary Instrumentation Framework to Profile Code Running on Intel GPUs", *28th IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, Singapore, May 2022.

[33] H. Theiling, "Extracting Safe and Precise Control Flow from Binaries", *7th International Conference on Real-time Systems and Appplications (RTCSA)*, Cheju Island, South Korea, December 2000.

[34] O. Villa, M. Stephenson, D. Nellans and S. W. Keckler. "NVBit: A Dynamic Binary Instrumentation Framework for NVIDIA GPUs", *52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, Columbus, Ohio, October 2019.

[35] S. B. Yadavalli and A. Smith, "Raising Binaries to LLVM IR with MCTOLL (WIP paper)", *20th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, and Tools and Theory of Embedded Systems (LCTES)*, Phoenix, Arizona, June 2019.