# UROBOROS: Instrumenting Stripped Binaries with Static Reassembling

Shuai Wang, Pei Wang, and Dinghao Wu
College of Information Sciences and Technology
The Pennsylvania State University
{szw175, pxw172, dwu}@ist.psu.edu

***Abstract*—Software instrumentation techniques are widely used in program analysis tasks such as program profiling, vulnerability discovering, and security-oriented transforming. In this paper, we present an instrumentation tool called UROBOROS, which supports static instrumentation on stripped binaries. Due to the lack of relocation and debug information, reverse engineering of stripped binaries is challenging. Compared with the previous work, UROBOROS can provide *complete*, *easy-to-use*, *transparent*, and *efficient* static instrumentation on stripped binaries. UROBOROS supports *complete* instrumentation by statically recovering the relocatable program (including both code and data sections) and the control flow structures from binary code. UROBOROS provides a rich API to access and manipulate different levels of the program structure. The instrumentation facilities of UROBOROS are *easy-to-use*, users with *no* binary rewriting and patching skills can directly manipulate stripped binaries to perform smooth program transformations. Distinguished from most instrumentation tools that need to patch the instrumentation code as new sections, UROBOROS can directly inline the instrumentation code into the disassembled program, which provides *transparent* instrumentation on stripped binaries. For efficiency, in the rewritten output of existing tools, frequent control transfers between the attached and original sections can incur a considerable performance penalty. However, the output from UROBOROS incurs *no* extra cost because the original and instrumentation code are connected by "*fall-through*" transfers. We perform comparative evaluations between UROBOROS and the state-of-the-art binary instrumentation tools, including DynInst and Pin. To demonstrate the versatility of UROBOROS, we also implement two real-world reengineering tasks which could be challenging for other instrumentation tools to accomplish. Our experimental results show that UROBOROS outperforms the existing binary instrumentation tools with better performance, lower labor cost, and a broader scope of applications.**

## I. INTRODUCTION

Software instrumentation inserts extra instructions to the target program to achieve multifaceted tasks. For example, instrumented code can record process run-time behavior [20], support program analysis [17], [22], or harden the executable layout to improve security [32], [34]. The instrumentation task can be performed at different stages: at compile time [21], at run time [22], or statically on executable files [34], [32]. In this paper, we present UROBOROS, a tool performing static instrumentation on stripped binaries.

The primary target of UROBOROS is stripped binaries, i.e., binaries with no debug or relocation information. In order to hinder reverse engineering and reduce the executable size, debug and relocation information is usually removed from binaries before releasing to the public. Without sufficient program information, reverse engineering on stripped binaries could be problematic and fragmentary. In particular, as the disassembled output of stripped binaries is *unrelocatable*, program transformations have to be very conservative for correctness, leading to enormous challenges for binary instrumentation. With several novel disassembling methods [29], UROBOROS recovers relocatability for stripped binaries. This enables painless program instrumentation because on relocatable assembly programs users can statically inline extra code to the instrumentation points. The instrumentation output, including both the original program and the instrumentation code, can be then assembled back into normal binaries. In sum, UROBOROS delivers complete, ease-to-use, efficient, and transparent instrumentation on stripped binaries.

*Complete.* Most existing work leverages dynamic analysis to recover relocation information, in which only incomplete functional components are obtained from the input [31], [13]. UROBOROS disassembles input binaries and recovers relocation information in both data and code sections through advanced static analysis. As the whole disassembled output is *relocatable*, program-wide transformations become feasible without common binary instrumentation drawbacks introduced by unrelocatable code or data snippets.

*Easy-to-use.* Existing static binary instrumentation tools need to rewrite the input binary. The instrumentation code is usually attached to the rewritten output to generate a workable executable [16], [4], [33], [12]. Although the rewriting process on stripped binaries could be complicated, tedious, and even problematic, the static instrumentation facilities of UROBOROS are make it easy, even for users with *no* binary rewriting skills. In fact, distinguished from existing static binary instrumentation approaches, UROBOROS does not need to rewrite the input binary. Legacy binaries are disassembled into *relocatable* programs, which can be instrumented as easily as compiler-generated assembly code. Users with *only* source code analysis and transformation skills can find no difficulty in using UROBOROS because they are essentially facing the same tasks they are familiar with.

*Transparent.* UROBOROS performs *reassembly-based instrumentation*. Previous static instrumentation tools rewrite the input binary to patch the instrumentation code or replicate the original code section into two and instrument both. In con-

trast, instrumentation code of UROBOROS is directly inlined into each instrumentation point. Transparent instrumentation enables UROBOROS itself or any other binary instrumentation tool to re-process the UROBOROS output easily. This feature can greatly broaden the application scope by bridging UROBOROS with existing infrastructures and potentially puts UROBOROS as the foundation of most binary analysis tasks.

*Efficient*. As jump instructions are frequently used to redirect control transfers between original code sections and patched sections, existing static binary instrumentation tools can incur relatively high execution slowdown and size increase on the rewritten output. However, as UROBOROS inlines instrumentation code at each instrumentation point, instrumentation code is connected with the context by "fall-through" transfers. In fact as jump instructions are not used anymore, UROBOROS engenders no additional execution cost from the instrumentation process, which is very efficient compared with previous binary instrumentation tools.

*Key Contributions and Results*

*1) Reassembly-based Instrumentation:* Different from the existing binary instrumentation techniques, UROBOROS delivers static reassembly-based instrumentation, i.e., the instrumentation output can be readily reassembled back to generate a binary again. UROBOROS recovers relocatable assembly code, enabling painless program-wide transformations. Due to the lack of relocation information, traditional static binary instrumentation (SBI) tools insert jump instructions at instrumentation targets to redirect control flow transfers. Note that the inserted jump instructions can bring in a non-negligible performance penalty in the instrumentation outputs. Furthermore, dynamic binary instrumentation (DBI) tools hook the target process and instrument the program during run time, which can lead to even higher execution cost. By contrast, benefiting from relocatability, no additional overhead is introduced by UROBOROS, which is very efficient compared with the existing SBI and DBI tools.

*2) An Easy-to-use Rich Instrumentation API:* UROBOROS translates the input binary into its internal representation and then recovers the program control flow structures on top of the representation. It provides an easy-to-use rich API to inspect and manipulate the internal representation and program structures. Currently, UROBOROS provides access to data bytes, instructions, basic blocks, functions, control flow graphs (CFGs) and call graphs (CGs). We demonstrate the versatility of the UROBOROS API by instrumenting input binaries at different levels of the program structure and also presenting two real-world instrumentation applications.

*3) Enable Novel Applications and Boost Existing Applications:* We present and evaluate two applications on top of UROBOROS. To our best knowledge, UROBOROS enables iterative software diversification on *stripped binaries*. On trace profiling, UROBOROS delivers significantly better instrumentation performance. We present an in-depth study of UROBOROS comparing with the industry-standard DBI tool, Pin [20]. We summarize our results as follows:

- *Iterative diversification.* A novel disassemble-diversify-reassemble workflow is enabled by UROBOROS. The diversified output is reprocessed for multiple times, boosting the diversification due to the "iteration effect". We observed notable binary similarity score decreasing with more iterations of processing.
- *Trace profiling on SPEC2006 and Linux common utilities.* UROBOROS incurs around 2.37X execution slowdown comparing with the native execution, while Pin imposes a sharp 8.83X slowdown on average.

*4) Scope and Limitations:* UROBOROS is mainly designed to process stripped binaries without debug or relocation information. Most challenges originate from the difficulties in precisely disassembling binary program and producing relocatable code. As a result, we assume input binaries are not obfuscated. In addition, we assume binaries to instrument do not dynamically generate code or feature self-modifying.[1] Binaries compiled from typical C programs fit these assumptions very well.

Our previous work proposes several methods to recover the relocation information from stripped binaries [29], some of which can have potential false positive and negative although a comprehensive study on large sets of program binaries shows that such incorrectness is extremely rare. Besides, one method relies on the information of function starting addresses, but recent research has made notable progress on this issue [8], [27]. UROBOROS leverages multiple strategies to recover the function information, and it can also be configured with user provided function information.

Currently, UROBOROS mainly takes binaries compiled from C code as the input. C++ programs can be processed by UROBOROS as long as it does not rely on the exception handling. Since binaries store the exception handling meta-data as separate sections, parsing these sections requires additional engineering efforts. We leave the complete support for C++ binaries as part of our future work.

By the time of writing, UROBOROS supports ELF binaries on both x86 and x64 architectures. UROBOROS makes no assumptions on what compilers are used to generate the input binaries, and it produces relocatable assembly code regardless of the input binaries being stripped or not. We believe that UROBOROS has initiated a new focus on binary reverse engineering and instrumentation by delivering reassembly-based instrumentation.

The rest of the paper is organized as follows. We first give an overview of the UROBOROS instrumentation capability in §II, and then present the design and implementation details in §III. We evaluate the UROBOROS instrumentation cost in §IV, and present two sample applications in §V. Finally, we review related work in §VI and conclude the paper in §VII.

[1]Note that UROBOROS and other static tools face similar challenges on correctly disassembling obfuscated binary code or self-modifying code [9], [30].

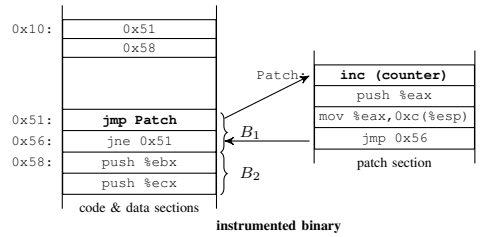


**Fig. 1:** Patch-based instrumentation.

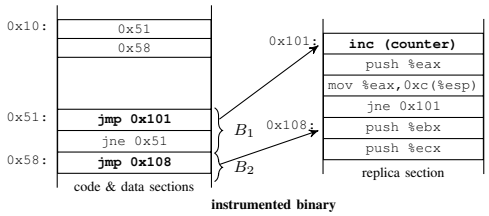


**Fig. 2:** Replica-based instrumentation.

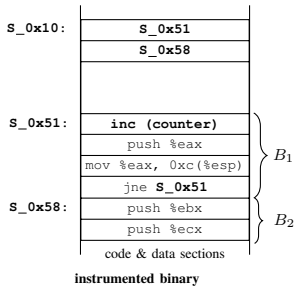


**Fig. 3:** Reassembly-based instrumentation.

## II. Static Binary Instrumentation

The UROBOROS API enables static instrumentation on stripped binaries in a program-wide scope. In this section, we first compare reassembly-based binary instrumentation with two commonly-used static binary instrumentation strategies used in the previous tools, i.e., patch-based instrumentation and replica-based instrumentation. We then demonstrate the UROBOROS instrumentation capability by creating an instrumentation application to trace memory writes.

### A. Reassembleable Instrumentation

As previous binary disassemblers only recover *unrelocatable* assembly programs, existing static instrumentation tools have to deliberately instrument input binaries without breaking memory references. To rebuild the instrumentation code and the input binary into a workable output, two rewriting methods are usually used, i.e., patch-based instrumentation and replica-based instrumentation. Instrumentation tools using the first strategy patch instrumentation code as a new section to the input binary. Jump instructions are inserted in the original code which redirect control transfers to the patched section. Replica-based instrumentation duplicates the original code sections into two; the replica is instrumented while jump instructions are inserted to the original which forward indirect control transfers to the replica. Both existing strategies could become challenging, and it may require users with specific rewriting skills to handle the whole process. In this section, we take a binary instrumentation task as an example to compare reassembly-based instrumentation with existing methods. Fig. 4 presents the layout of a stripped binary. Note that memory references (e.g., 0x51) are all *unrelocatable* immediate values. The code section contains two basic blocks ($B_1$, $B_2$); suppose we want to instrument basic block $B_1$ to add a counter instruction at the beginning.

*1) Patch-based Instrumentation:* Patch-based instrumentation replaces instructions at the instrumentation point with a jump instruction, which points to a new section patched at the end of the input binary. The newly added section contains both the instrumentation code and the replaced instructions. Fig. 1 presents the code layout after patching. As a long jump (e.g., "`jmp Patch`") needs to occupy 5 bytes, two instructions are relocated to leave enough space. During run time, the inserted

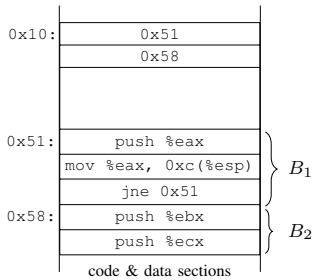


**Fig. 4:** An example of instrumentation target.

jump instruction redirects the control flow to the patched section, and after executing the instructions on the patched section, another jump instruction redirects control flow back to the original code section. As two control transfers could occur at each instrumentation point, this rewriting strategy introduces relatively high performance slowdown. Even more seriously, usually more than one instruction have to be replaced, and the replaced instructions are required to be *relocatable* in order to keep the correct semantics. It is not always obvious whether these instructions can be safely relocated. Optimization techniques are proposed to use a short jump or even an interrupt (`int 3`) to transfer control to the patched section, but short jump (2 bytes on x86 architecture) still cannot substitute a single byte instruction and frequent interrupt handlings have a big penalty on the execution [19], [23].

*2) Replica-based Instrumentation:* Typical instrumentation tools in this category generate a replica of the original code section. The replica contains both the instrumentation and the original code. As the replica has a different range of memory addresses, memory references in the original code, e.g., destinations of function calls, are cautiously rewritten with new memory addresses in the replica at the best effort. Jump instructions are deliberately inserted at control flow destinations in the original copy, forwarding indirect memory references to their new targets in the replica in case some address translations are missed and the execution control is transferred to the original copy. This replica-based instrumentation can reduce performance slowdown caused by frequent control transfers in the instrumentation output, especially when there are large amounts of instrumentation targets, e.g., all the

basic blocks. However, the size of rewritten output can be notably increased.

In general, the rewritten output by either patch-based or replica-based instrumentation changes the internal structure of input binaries, and it becomes very challenging or even impossible to apply binary analysis and transformation on the instrumented binaries. Our experiments in §IV report that disassembling the rewritten outputs can lead to many decoding errors, while no error is found when disassembling their original inputs. Deng, Zhang, and Xu [15] also discuss how patch-based and replica-based instrumentations can impede binary component extraction and embedding. Overall, in addition to high cost imposed on the instrumentation output, existing instrumentation tools cannot undertake *transparent* instrumentation, which narrows their application scope as well.

*3) Reassembly-based Instrumentation:* As UROBOROS can recover an executable in a *relocatable* format before instrumentation, the instrumentation code can be directly *inlined* into the target. The instrumentation output is then assembled to produce a *normal* binary output. As shown in Fig. 3, memory references in the unrelocatable program have been translated into relocatable formats (e.g., "`S_0x01`" and "`S_0x51`"), and UROBOROS directly inserts the counter instruction at the instrumentation point. As the counter instruction is inlined in the context, instrumentation cost introduced by frequent control transfers or replicated code is indeed avoided. Moreover, given all the memory references in a relocatable format, linkers will resolve these references with new memory addresses during reassembling. Code pointers can naturally refer to their original destinations at this time, and no intentional binary rewriting is needed to adjust the value of code pointers. In addition, the instrumentation output can be seamlessly reprocessed by further binary analysis or transformation without any particular difficulty introduced by the UROBOROS instrumentation.

### B. UROBOROS *Kits*

In Fig. 5, we present the code that a user needs to write if he wants to trace memory writes. The code is written in the OCaml programming language. The `process` function (line 20) utilizes the UROBOROS `Instr_visitor` module to traverse the whole instruction list and iterate all the memory writing instructions. By leveraging the `is_mem_write` function (line 23) from the `Instr_utils` module, memory writing instructions are filtered out during iterating. Users only need to define their `visit` function (line 22), and UROBOROS takes care of the underlying details. Memory writing instructions are classified into three categories according to the number of operands each instruction has (line 8). The `Instr_template` module provides the `gen_logging_instrs` function (line 14) to generate logging instructions, which initializes a sequence of instructions for logging according to the target instruction and its location information. Finally, the `insert_instr_list` function (line 24) from the `Instr_utils` module updates programs

```
1  open Type
2  open Utils
3  open Instr_utils
4  let il_update = ref []
5
6  let instrument i t =
7    let module IT = Instr_template in
8    match t with
9      | SINGLE_WRITE
10     | DOUBLE_WRITE
11     | TRIPLE_WRITE ->
12       il_update :=
13       (get_loc i
14       |> IT.gen_logging_instrs i) @ !il_update;
15   (* relocation labels on instrumentation target have
16    * been given to inserted code; remove redundancy *)
17       eliminate_label i
18     |_ -> i
19
20 let process il =
21   let module IV = Instr_visitor in
22   let visit i t = instrument i t in
23   IV.map_instr is_mem_write visit il
24   |> insert_instr_list BEFORE !il_update
```

**Fig. 5:** A UROBOROS plugin for tracing memory writes.

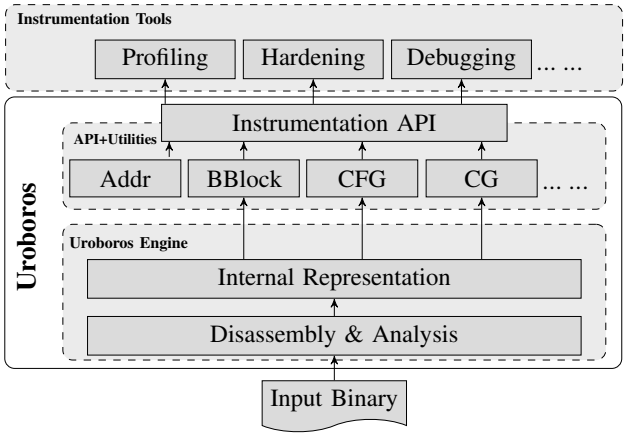


**Fig. 6:** The architecture of UROBOROS.

by inserting instrumentation code at instrumentation points. This function supports instrumentation before or after target instructions.

By providing sufficient instrumentation facilities, UROBOROS only exposes high-level interfaces to users. In the above example, we visit programs at the instruction level and search for memory write operations. The `Instr_utils` module provides various instruction filtering utilities (e.g., `is_mem_write`, `is_jmp`) to help users classify instructions according to the functionality. UROBOROS also provides a rich API to traverse and access multiple levels of the program structure. More examples are presented in §V.

## III. DESIGN

### A. Overview

The design overview is shown in Fig. 6. UROBOROS consists of three main modules—the disassembly module, the analysis module, and the instrumentation module. The

disassembly module takes executable files as inputs and recovers unrelocatable disassembled outputs. The disassembly module also dumps the data and metadata sections from the input. The analysis module identifies memory addresses from all the immediate values found in the input and lifts them into relocatable symbols. The analysis module also recovers control flow structures on top of the relocatable program. The recovered program is parsed into multiple levels of internal representations, and the instrumentation module provides utilities to access and manipulate these internal data structures. Users can utilize the provided facilities to undertake program-wide analysis and transformations on the relocatable program. As we have emphasized, the instrumented output of UROBOROS can be directly assembled back into a normal binary, which can even be re-processed by UROBOROS for multiple times.

The disassembly module of UROBOROS implements the disassembling algorithm proposed in BinCFI [34]. We consider data bytes embedded in code sections the main reason to make linear disassembling fail. In this algorithm, the linear disassembler leverages a validator to correct disassembling errors due to the embedded data bytes. As suggested by BinCFI, we rely on three rules to validate the raw disassembling results and locate "data gaps", i.e., invalid opcode, direct control transfers outside the current module, and direct control transfer to the middle of an instruction. According to these rules, we implement a multiple-round disassembling validation process.

Various kinds of relocatable symbols exist in compiler generated assembly programs, e.g., jump table entries, function pointers. During the link time, linker substitutes each relocatable symbol with a concrete value, which represents the runtime memory address of the corresponding object . To make the disassembled output relocatable, the main challenge is to identify memory addresses from all the data found in the disassembled output. The identified memory addresses can be then translated into relocatable symbols that enable the reassembly-based instrumentation. However, it is indeed quite challenging to find memory addresses in the disassembled output, as theoretically it is impossible to decide statically any suspicious immediate value as a memory address or some random data without execution. We leverage multiple methods to guide UROBOROS in identifying memory addresses from immediate values. These methods are proposed based on observations of the disassembled outputs from real-world programs [29]. An in-depth study shows that with all the methods applied, almost no false positive or negative is found on broad sets of real world binaries.

### B. Instrumentation Module

*1) Internal Data Representation:* UROBOROS supports program-wide instrumentation by recovering control flow structures of the input program. The disassembled instructions, dumped data sections, and metadata are parsed into an intermediate representation for analysis and transformations. We now introduce how we organize these internal representations.

We have created a hierarchical representation for the recovered program to support program-wide analysis and transformation. The topmost level in this hierarchy is the *Module* object, which corresponds to each disassembled binary. This object stores the binary-level information of inputs. Typically program binary stores its data, code, and metadata in different binary sections, and UROBOROS keeps each section in one object. Control structure units, i.e., basic blocks and functions, are stored in their corresponding *BBlock* and *Function* objects. UROBOROS also maintains *Instruction* objects, which represent instructions in the code section. Three data sections (`.rodata`, `.data`, and `.bss`) are stored inside UROBOROS, and UROBOROS uses one object to represent each data byte. Metadata from the input binaries is stored in the same way. Furthermore, each memory address is maintained in an `Addr` object. `Addr` objects maintain both memory labels and memory addresses. All the internal modules have their associated `Addr` objects. Labels and memory addresses can be updated to support arbitrary manipulations that may change memory addresses in the transformed output.

*2) Support Binary Instrumentation:* Stripped binaries usually do not contain sufficient control flow information, e.g., functions, basic blocks, or control flow graphs. This information is recovered in the UROBOROS analysis module. UROBOROS recovers function entry point addresses by collecting destinations in `call` instructions and exported function information from the symbol table.[2] UROBOROS also identifies functions by matching instructions with typical function starting patterns; 52 instruction patterns are implemented in this step. In addition, UROBOROS can be configured with user provided function entry point addresses.

UROBOROS recovers basic blocks as the units of the control flow structure. Basic blocks are identified in its standard way, i.e., instruction sequences with only one control flow entry point and one exit point. We collect all the identified code pointers and control flow instructions to divide instructions into multiple blocks. Note that as the destinations of most indirect jump instructions are code pointers that can be found in code and data sections, UROBOROS is capable of identifying basic blocks determined by indirect jumps according to the collected pointers. Each basic block object maintains information of its predecessors and successors on the CFG. For indirect control flow transfers, we conservatively consider they can transfer to the beginning of any basic block.

UROBOROS and other static instrumentation tools face similar challenges (undecidability issues), for example, basic block and function recognition [8], [27]. We are currently working on more sophisticated strategies to recover function boundaries, and these techniques will be merged into the UROBOROS future releases. Besides, the identification of indirect control transfer destinations can be improved through value-set analysis (VSA) [7], which is left for our future work.

[2]Note that a symbol table does often exist in stripped binaries that contains, for example, "exported function" information to support dynamic linkage.

In general, 21 modules, including 121 functions, are implemented in UROBOROS to support binary instrumentation. These utilities include common operations at different levels of the program structure (e.g., function, basic block and instruction). For instance, UROBOROS provides multiple visitor modules to support visiting different levels of the program structure in a flexible way, e.g., `Function_visitor`, `BBlock_visitor`, and `Instr_visitor`. Utility modules are also provided to query, modify, and remove internal objects. For example, UROBOROS provides `Instr_utils` and `BB_utils` modules to support inspection and manipulation on `Instruction` and `BBlock` objects. In addition, the `Instr_utils` module provides functions to query instruction types, for example, `is_mem_write`, `is_mem_read`, `is_call`, `get_label`, and `get_addr`. If modules (*Instruction*, for instance) are maintained in a list, UROBOROS also provides utilities to traverse all the entries on the list one by one.

*3) Instrumentation on Assembly Code:* Currently UROBOROS supports inserting instrumentation instructions in the format of its internal representation. Instrumentation code can be inserted as follows:

```
insert_single_instr pos loc i_insert il
```

This function inserts instrumentation instructions at the given location. `Pos` is a predefined type, including both `BEFORE` and `AFTER` type variables. Instrumentation code can be inlined before or after the target instruction. `BEFORE` means inserting instrumentation code in front of the target, while `AFTER` means behind. `i_insert` represents the assembly instructions in the UROBOROS internal data structures. `loc` consists both memory address and possible relocation labels referring to that address. `il` is the internal representation of input program instructions.

As shown in Fig. 5, another frequently-used function is `insert_instr_list`. In case a large amount of instrumentation code needs to be inserted at different places, users need to construct each instrumentation instruction with its desired instrumentation position. The instrumentation instruction list is then provided to this function, as well as the `pos` type. This function sorts the instrumentation code according to their desired memory positions and inlines instrumentation instructions into the targets.

Besides instrumentation code insertion, we also provide functions to replace existing instructions with instrumentation code as follows:

```
sub_single_instr i_t i_sub il
```

This function uses instrumentation code to replace the given target. Here, `i_t` and `i_sub` represent the target instruction and its substitution while `il` stands for the whole instruction list. UROBOROS also provides function `instrument_update` to support instrumentation code updating in both *insertion* and *substitution* operations. This function requires users to construct bundles which consist of instrumentation code and the associated instrumentation types, i.e., `INSERT` or `SUB`.

In addition, UROBOROS can output the text representation of assembly code which can further facilitate ad-hoc or customized analysis and transformation.

*4) CPU Flags Usage Optimization:* Many instrumentation scenarios require the instrumentation code to have a counter; whenever execution flow hits the instrumentation point, the counter is incremented by a fixed stride. The maintained counter can be used to record the number of executed instructions, or as the index of an array to record some execution information. However, opcodes (`inc`, for instance) that are usually used to increment the counter need to change CPU flags, and in order to preserve the correct semantics, CPU flags need to be saved and restored for instrumentation. Opcodes `pushf` and `popf` are designed to save and restore CPU flags on the stack. However, both opcodes can delay the instruction pipelining and cause relatively high performance slowdown.[3] In fact, some of our early experiments show that performance slowdown could reach almost 700% for a trace profiling instrumentation on `gzip`.

Instead, UROBOROS deliberately selects instructions to optimize the flag manipulation operations. Templates are provided to users so that they can easily construct optimized instrumentation code. For the commonly used counter-increment scenario, UROBOROS can even avoid changing CPU flags by misusing the `loop` opcode.[4] `Loop` does not change any flag and decreases the value in register `ecx` each time until it reaches zero. A classic example is shown below, which records the number of executed basic blocks.

```
BB:      push %ecx
         movl index, %ecx
         loop BB_stub
BB_stub: movl %ecx, index
         pop  %ecx
```

On 32-bit x86 architecture, the global variable `index` needs to be initialized with `0xffffffff` before execution, and the total number of executed basic blocks can be calculated by subtracting `0xffffffff` with the final value in `index`.

Another optimization is to use opcode `lahf` and `sahf` to speed up the saving and restoring of CPU flags. However, tests show that `lahf` and `sahf` slow down the execution for about 15% compared with the "loop" optimization. Also these two opcodes are absent for early AMD and Intel CPUs. UROBOROS still provides templates using these opcodes to construct instrumentation code for the sake of handling more general cases. The accessibility of low-level details makes UROBOROS quite flexible when facing some cost-sensitive instrumentation scenarios.

[3]The instruction `pushf` takes 3 uops while `popf` takes 9 uops on the Intel Haswell architecture.

[4]We learned this optimization from a discussion at http://goo.gl/Fb0Djk. The trick was initially suggested by Guntram Blohm.

## IV. Evaluation

UROBOROS instrumentation is directly applied to the relocatable assembly, and only negligible cost is introduced without actually inserting instrumentation code. The experiments have shown a trivial instrumentation, i.e., disassembling a binary and reassembling it as what it was, leads to at most 1% execution slowdown and size expansion, when evaluating a broad set of real world program binaries. We do not report the detailed results of this trivial evaluation, for the data is not very informative. Instead, we compare UROBOROS with another static instrumentation tool DynInst [12], [16] with respect to the performance and size of instrumented binaries. We also evaluate the execution time of UROBOROS itself to demonstrate its efficiency. We set up all the experiments on a server machine with a 2.90GHz Xeon(R) E5-2690 CPU and 128GB RAM.

We pick DynInst (version 8.2.1, http://www.dyninst.org/), a widely used static binary instrumentation framework, as the competitor of UROBOROS. DynInst features both patch-based and replica-based instrumentation. For instrumentation tasks with small amount of instrumentation targets, DynInst undertakes a patch-based instrumentation, in which two control transfers occur before and after executing instrumentation code. However, patch-based instrumentation could result in many additional control transfers when instrumenting large amount of targets (e.g., instrumentations on every basic block or function). In that case, DynInst will switch to the replica-based instrumentation to reduce the execution overhead at the expense of code size bloating.

We use basic block and function counting instrumentation as the benchmark to evaluate both tools on all the C programs from 32-bit SPEC2006. We record execution slowdown and size increase for the instrumented binaries. We compile all the test cases from source code with the default configurations. All the test cases are stripped before processing by UROBOROS (with the `strip` tool from GNU Binutils).[5] Before the cost evaluation, we first verify the functionality of the instrumentation output with test cases officially provided by SPEC2006. Verification shows that *all* binaries instrumented by UROBOROS successfully pass all test cases, while two binaries (`gcc` and `mcf`) processed by DynInst fail the functionality testing.

Fig. 7 presents the size increase for the function and the basic block level instrumentation. For both tests, DynInst increases binary sizes to more than twice of the original (131.2% increase for basic block counting and 119.1% for function counting), while UROBOROS only brings in less than 40% of size expansion for basic block counting and 2.0% for function counting. Note that for several cases, UROBOROS only introduces negligible size increase (`lbm` in both evaluations; `mcf` and `bzip` in function counting evaluation). It is worth mentioning that the instrumentation output of DynInst will load both the original and replica code into memory during run time. UROBOROS has a quite big advantage in terms of memory efficiency.

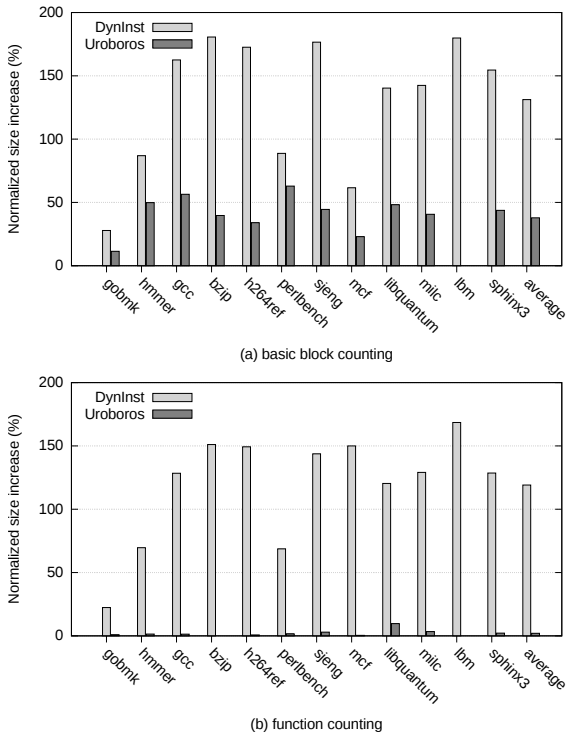


**Fig. 7:** Size increase comparison.

Fig. 8 presents the performance slowdown for the function and basic block level instrumentation. As DynInst enables the "replica-based" instrumentation, we expect that execution slowdown of DynInst could compete with the UROBOROS reassembly-based instrumentation. Indeed, for function level instrumentation, the average execution slowdown of DynInst (2.94%) is as good as UROBOROS (2.93%). For the basic block counting task, binaries instrumented by UROBOROS is slower than DynInst (93.38% compared to 76.56%). We attribute this to that DynInst adopts multiple customized optimization solutions. One such example is liveness analysis. The `inc` instruction used for counting basic blocks modifies CPU flags. In this experiment, we conservatively save/restore the flags in each instrumentation process. DynInst, on the other hand, performs liveness analysis to avoid some saving/restoring operations, offering an enhanced performance gain.

Still, we consider around 93% instrumentation slowdown is acceptable for instrumentation tasks with many targets such as basic block counting. On the other hand, as we reported, all the instrumented binaries of UROBOROS preserve the correct semantics, while two instrumented cases of DynInst (`gcc` and `mcf`) do not pass the functionality testing. The reason may because the DynInst instrumentation facilities contain certain bugs, or the employed rewriting strategy is likely to break the correctness when instrumenting complex binaries.

[5]Similar to other tools, UROBOROS cannot fully recognize all the functions in stripped binary code. We assume the function information is known to us in §IV and §V.

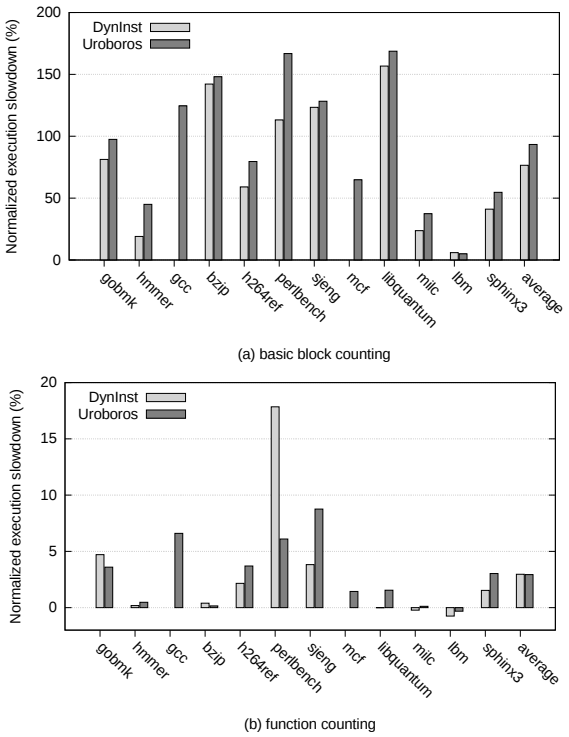


**Fig. 8:** Performance slowdown comparison.

As aforementioned, previous instrumentation techniques significantly deforms binary structure, inevitably complicating subsequent analyses and transformations. In this experiment, we found that the commonly-used disassembler `objdump` reports a considerable number of decoding errors on the output from DynInst. In the UROBOROS case, the instrumented binaries can be successfully disassembled by `objdump` with no error.

We also evaluate the processing time of UROBOROS with or without instrumentations. We design three tasks for evaluation, each of which instruments binaries at different levels of the program structure. We evaluate UROBOROS against the same set programs from 32-bit SPEC2006. The processing time is calculated from starting to disassemble the inputs until finishing assembling the instrumentation outputs.

We evaluate UROBOROS in terms of instruction, basic block, and function level instrumentations. Our instruction level instrumentation inserts "sandbox" instructions to all the indirect control transfers. An `and` instruction is used to validate the destinations of the upcoming control transfers. This instrumentation is quite useful in developing goal-oriented security applications, e.g., software fault isolation (SFI). The basic block and function level instrumentation inserts instructions at the beginning of each basic block or function to count the number of executed units.

Table I presents the evaluation data. The "baseline" column presents processing time without applying instrumentation while the other three columns show data with instrumentations. "Instr. A" corresponds to instruction level instrumentation;

**TABLE I:** Instrumentation time evaluation.

| Program | Baseline (s) | Instr. A (s) | Instr. B (s) | Instr. C (s) |
|---|---|---|---|---|
| bzip | 0.72 | 0.77 | 0.87 | 0.77 |
| hmmer | 3.04 | 3.11 | 3.64 | 3.11 |
| gcc | 67.93 | 72.46 | 77.37 | 69.24 |
| gobmk | 31.75 | 32.04 | 35.89 | 32.39 |
| h264ref | 5.91 | 6.02 | 6.74 | 6.00 |
| perlbench | 12.92 | 13.04 | 16.43 | 13.08 |
| sjeng | 8.87 | 8.91 | 9.12 | 8.93 |
| mcf | 0.39 | 0.40 | 0.42 | 0.40 |
| libquantum | 0.62 | 0.62 | 0.69 | 0.63 |
| milc | 1.31 | 1.32 | 1.52 | 1.34 |
| lbm | 0.37 | 0.38 | 0.43 | 0.40 |
| sphinx3 | 1.91 | 1.91 | 2.26 | 1.94 |
| average | 11.31 | 11.75 | 12.95 | 11.52 |

"Instr. B" and "Instr. C" represent basic block and function level instrumentations, respectively. On average, the instruction level instrumentation increases the processing time by 3.89%, basic block level instrumentation increases 14.50% while function level instrumentation increases 1.86%. The basic block level instrumentation has the most candidates to instrument, and therefore it imposes the highest instrumentation time cost.[6] Overall, the instrumentation processing time of UROBOROS is quite small.

## V. SAMPLE APPLICATIONS

To demonstrate how UROBOROS can be used in practice, we developed two applications for UROBOROS. The first application, UroborosDiv, instruments the input binary for a large amount of iterations. An unique "iteration effect" is leveraged to boost the generation of large amounts of diversified binaries with increasing performance. The second application, UroborosTrace, records executed basic block information. It can be used for trace profiling. By demonstrating these applications, we show UROBOROS can perform some transformations that existing tools can hardly handle. For some of the transformations that can be done by other tools, we show that UROBOROS can result in better performance.

### A. Software Diversification

Although it is originally proposed for optimization, function inlining has been employed as a software diversification strategy [18]. To demonstrate the strength of UROBOROS in binary instrumentation and transformation, we present a software diversification application UroborosDiv in this section. UroborosDiv is an instrumentation application that diversifies binaries by randomly selecting functions from a set of candidates and inlining them into their call sites. Different from the "one-off" design of most diversification frameworks, in which the original input is used to generate diversified copy one, two, three, etc., UroborosDiv takes the instrumentation output as the input and re-instrument it iteratively. As discussed in §II-A, existing instrumentation tools, either patch-based or replica-based, can hardly process binaries in this iterative manner because massive binary structure changes impede secondary

[6]Note that instruction level instrumentation only targets *indirect control transfer* instructions, while basic block level instrumentation targets *all* the basic blocks.

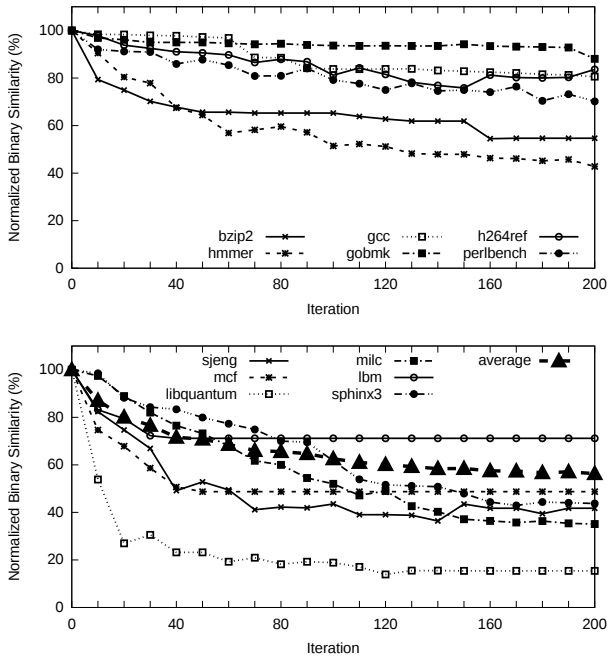


**Fig. 9:** Binary similarity rates under different iterations.

instrumentation. However, transformation through UROBOROS is *transparent*, i.e., UROBOROS produces *normal* binaries in which all newly inserted code is blended into the original code. Therefore, instrumented binaries can be directly re-processed by any existing binary tool, including UROBOROS itself.

We apply UroborosDiv to all the SPEC2006 C programs. By iterating the instrumentation procedure on a program binary, we can quickly produce a large number of unique copies. In the experiments, we iterate the pass for 200 times. After processing, we use test cases from 32-bit SPEC2006 to verify the functionality correctness on the diversified binaries, and all diversification outputs pass the functionality testing.

We evaluate the diversification effect by using the industry standard binary diffing tool BinDiff.[7] For each diversification output, we diff it with the original input and get the instruction level matching rate.[8] The evaluation data for all tested programs are shown in Fig. 9. Binary similarity rate decreases for all the test cases. Note that as one function is inlined at each iteration, binaries with larger code sections, e.g., `gobmk`, `gcc`, get higher matching rates in general. It should be clarified that when this application is deployed in real-world scenarios, more iterations can be applied, also with more functions to be inlined in each iteration.

We have noticed a "stabling" trend in the case of `lbm`, `mcf` and `libquantum`. These binaries have relatively fewer functions, and it is likely that UroborosDiv has consumed all inline-able targets after certain iterations. Overall, as shown in the "average" data of Fig. 9, average similarity rate decreases with more iterations. We interpret it as a promising result to show the diversification effect increases with more iterations. Again, the transparent instrumentation in UROBOROS allows us to re-process the instrumentation output for many iterations. A large amount of instrumentation outputs can be generated, with the distinguished diversification effect enabled by iterations.

[7] http://www.zynamics.com/bindiff.html

[8] BinDiff provides the number of matched functions, basic blocks and instructions. Since function and basic block can be "partially" matched, e.g., 30% of the instructions in a function are matched with another function, counting the number of matched functions or basic blocks could be tricky. Therefore, we only adopt the instruction level matching rate.

### B. Trace Profiling

In this section, we demonstrate the efficiency of UROBOROS by comparing it with Pin [20] (version 2.14, build 71313), on trace profiling, a widely-used instrumentation task. We iterate the basic blocks of a binary, and instrument each basic block with a sequence of instructions for logging. We record executed basic blocks by writing their memory addresses in the original binary to a global buffer. As for Pin, we write a simple Pintool (a tool built on top of Pin) to instrument basic blocks on the execution traces in the same way. We strictly follow the optimal Pintool writing strategies for better performance, such as Pin code inline, fast-call linkage, and `IPOINT_ANYWHERE` to schedule the call anywhere. Note that the profile data is not flushed to the disk, and the buffer will be rewritten when it is full. We consider the measurement without I/O overhead can give us a better estimation of the instrumentation cost. As discussed in §III-B4, we avoid to save and restore CPU flags by misusing the `loop` opcode. Currently we allocate a 16M buffer to store the memory addresses of the executed basic blocks. We write a 4-byte memory address into the allocated buffer for each basic block.

Fig. 10 shows the performance overhead of running 32-bit SPEC2006 C programs. We use the shipped test cases to measure performance. For each program, we instrument the binary with trace logging code and present the data in "UROBOROS-Logging" bar. The "Null-Pin" bar presents the Pin environment overhead, which runs the binary under Pin without any instrumentation. The "Pin-Logging" bar shows overhead when Pin is executed with our basic block instrumentation Pintool. On average, "UROBOROS-Logging" incurs 2.77X performance overhead comparing with the native execution, while "Pin-Logging" imposes as much as 9.76X overhead. In fact, the overhead of UROBOROS-Logging is close to that of "Null-Pin", which is 2.47X. Note that "Pin-Logging" can generate over 15X performance penalty for some test cases, e.g., `gcc`, `milc`, and `sphinx3`, while the overhead of "UROBOROS-Logging" is relatively stable at around 2–4X. In fact, the average performance overhead of "UROBOROS-Logging" reaches the theoretically lowest value. Particularly, as on average a basic block has 5–10 instructions, and for each basic block, UroborosTrace inserts 7 new instructions, the total amount of executed instructions could double. Furthermore, as the `loop` instruction has a relatively high CPU ticks (7 for our experiment platform), we estimate the theoretical lowest

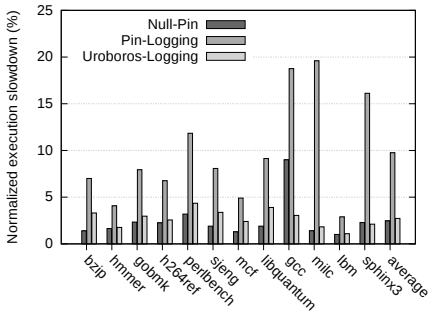


**Fig. 10:** Performance overhead comparison for SPEC2006.

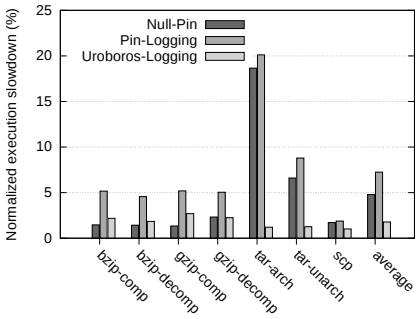


**Fig. 11:** Performance overhead comparison for common utils.

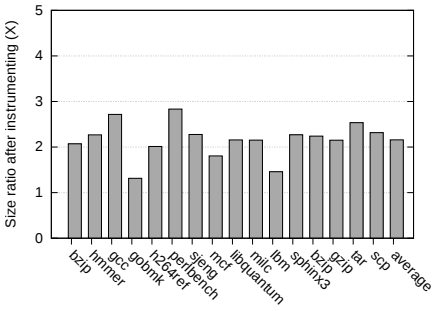


**Fig. 12:** Size increase after instrumentation.

execution slowdown is around 2.3–2.7X, which matches very well with "UROBOROS-Logging" (2.77X).

We also evaluate UroborosTrace on four common Linux utilities which represent three kinds of workload. The program `tar` is I/O bound, `bzip2` and `gzip` are CPU intensive program, and `scp` is a network application. We use `tar` to archive and extract the GNU Core utilities 8.13 package (about 50MB). The same input is used by two compressors for compressing and decompressing and `scp` sends the archived package through a 1 Gbps Ethernet link. Fig. 11 shows the evaluation data. The CPU bound programs show similar evaluation results with Fig. 10. UroborosTrace imposes relatively very low overhead on two tests of the `tar` program (on average 1.23X), while on average over 14.5X overhead is incurred by Pintool. Note that different from Fig. 10, the average data of "UROBOROS-Logging" is even better than "Null-Pin". Overall, UroborosTrace exhibits a decent runtime performance.

Fig. 12 presents the size increase of instrumented binaries. The average size increase is 2.16X, which matches well with our expectation. In summary, UROBOROS is quite *efficient* compared with real-world dynamic instrumentation tools.

## VI. RELATED WORK

The most closely related work to UROBOROS is binary instrumentation, which comes in two flavors: static and dynamic. Static approaches insert additional code to an executable and produce a new modified executable. Dynamic methods, in contrast, perform the instrumentation during run time without any modifications to the executable file. UROBOROS belongs to the category of the static binary instrumentation. Both of these two instrumentation approaches have their pros and cons. We also briefly introduce the related work on binary analysis and reverse engineering.

*1) Static Binary Instrumentation:* Static methods instrument the whole input binary before execution [4], [26], [14], [33], [19]. It has been widely used in security hardening tasks such as control-flow hijacking mitigation [30], software control-flow integrity enforcement [34], [32], and retrofitting security defenses [25]. However, due to the difficulties of disassembly, most previous static binary rewriting tools have to require relocation or debug information [14], [26], [19]. SecondWrite [4] performs advanced static analysis to lift binary code into LLVM IR. The IR is then employed for binary rewriting. PSI [33] supports robust security-related binary instrumentation through binary rewriting.

As we have pointed out, a common feature of conventional static binary instrumentation is that it relies on binary rewriting. It has to carefully relocate instructions at the instrumentation point to arrange space for newly inserted code. To this end, patch-based and replica-based instrumentations are frequently used. However, the newly generated binary could exhibit high execution slowdown, size increase, and even error functionality. The key feature of UROBOROS is that it does not rely on binary rewriting. Instead, it leverages the advanced disassembling technique [29] to directly inline the instrumentation code into the target binary. Therefore, UROBOROS delivers a decent runtime performance and a small increase in code size.

*2) Dynamic Binary Instrumentation:* Dynamic binary instrumentation inserts additional code when a program executes, which is more accurate than static binary instrumentation since it only considers the real path taken at run time [20], [10], [24]. Dynamic binary instrumentation has been widely used for program performance profiling [35] and security-oriented execution monitoring tasks [17], [22]. Pin [20] and DynamoRIO [10] undertake lightweight instrumentation jobs, while Valgrind [24] is designed for more heavyweight instrumentation tasks, e.g., memory debugging. Among them, Pin is widely used for goal-driven binary security tasks, such as dynamic taint analysis [17], [22]. DynInst [12], [16] supports both static and dynamic binary instrumentation. It disassembles the stripped binaries and instruments them statically or dynamically. Dynamic instrumentation methods cannot be deployed in some scenarios such as real-time or mission-critical systems due to the runtime instrumentation environment.

*3) Binary Analysis:* UROBOROS relies on advanced binary analysis to lift relocatable symbols and recover control flow structures. The previous work also provides multiple binary analysis functionalities [11], [28]. Besides, a set of advanced analysis techniques has been developed on top of them, such as

symbolic execution and fuzzy testing [5], [6]. Our work differs from these platforms in that we have different design goals. Their primary goal is saving binary analysis from tedious manual work while UROBOROS focuses on reassembly-based binary instrumentation.

*4) Reverse Engineering:* Most of the existing reverse engineering work does not produce re-compilable code [1], [2], [3]. They typically focus on generating code for the analysis purpose. Our previous work presents reassembleable disassembling, a new technique to recover the relocation information for reassembling [29], while in this research we extend it with a more general focus, i.e., reassembly-based static binary instrumentation. The main focus of this paper is to present the development of a general-purpose platform for reassembly-based static binary instrumentation. We show the versatility and performance of UROBOROS by elaborating the design of the UROBOROS API and infrastructure; we also evaluate UROBOROS with multiple binary instrumentation tasks. In contrast, our previous work mainly discusses how to recover program relocation information from stripped binaries. Another important contribution of this paper is a thorough comparative evaluation with the state-of-the-art static and dynamic binary instrumentation platforms.

## VII. CONCLUSION

In this paper, we present UROBOROS, the first framework that supports static *reassembly-based binary instrumentation*. UROBOROS recovers assembly program in a *relocatable* format, and delivers *complete*, *easy-to-use*, *transparent*, and *efficient* instrumentation. UROBOROS provides a rich API and utilities to support analysis and transformations on program internal representations and control flow structures. We evaluate UROBOROS by comparing it with the state-of-the-art static instrumentation tool. We also illustrate the versatility of the UROBOROS instrumentation facilities by developing two instrumentation applications. Evaluation results show that UROBOROS outperforms the existing tools in terms of lower cost instrumentation and more flexible applications.

## VIII. AVAILABILITY

The source code and documentation of UROBOROS are released publicly at https://github.com/s3team/uroboros.

## ACKNOWLEDGMENTS

We thank Jiang Ming for some helpful discussion and proofreading. This research was supported in part by the National Science Foundation (NSF) grants CNS-1223710 and CCF-1320605, and the Office of Naval Research (ONR) grant N00014-13-1-0175.

## REFERENCES

[1] "Dagger," http://dagger.repzret.org/.
[2] "MC-Semantics," https://github.com/trailofbits/mcsema.
[3] "The IDA Pro disassembler," https://www.hex-rays.com/products/ida/index.shtml.
[4] 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 *Proceedings of the 8th ACM European Conference on Computer Systems (EuroSys '13)*, 2013.
[5] T. Avgerinos, S. K. Cha, A. Rebert, E. J. Schwartz, M. Woo, and D. Brumley, "Automatic exploit generation," *Communications of the ACM*, vol. 57, no. 2, pp. 74–84, Feb. 2014.
[6] T. Avgerinos, A. Rebert, S. K. Cha, and D. Brumley, "Enhancing symbolic execution with veritesting," in *Proceedings of the 36th International Conference on Software Engineering (ICSE '14)*, 2014.
[7] G. Balakrishnan, "WYSINWYX: What you see is not what you execute," Ph.D. dissertation, University of Wisconsin-Madison, 2007.
[8] T. Bao, J. Burket, M. Woo, R. Turner, and D. Brumley, "ByteWeight: Learning to recognize functions in binary code," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security '14)*, 2014.
[9] G. Bonfante, J. Fernandez, J.-Y. Marion, B. Rouxel, F. Sabatier, and A. Thierry, "CoDisasm: Medium scale concatic disassembly of self-modifying binaries with overlapping instructions," in *Proceedings of the 22nd ACM Conference on Computer and Communications Security (CCS '15)*, 2015.
[10] D. L. Bruening, "Efficient, transparent, and comprehensive runtime code manipulation," Ph.D. dissertation, Massachusetts Institute of Technology, 2004.
[11] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification (CAV '11)*, 2011.
[12] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. of High Performance Computing Applications*, vol. 14, no. 4, pp. 317–329, 2000.
[13] J. Caballero, N. M. Johnson, S. McCamant, and D. Song, "Binary code extraction and interface identification for security applications," in *Proceedings of the 2010 Network and Distributed System Security Symposium (NDSS '10)*, 2010.
[14] 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, pp. 882–945, Sep. 2005.
[15] Z. Deng, X. Zhang, and D. Xu, "BISTRO: Binary component extraction and embedding for software security applications," in *Proceedings of the 18th European Symposium on Research in Computer Security (ESORICS '13)*, 2013.
[16] L. C. Harris and B. P. Miller, "Practical analysis of stripped binary code," *ACM SIGARCH Computer Architecture News*, vol. 33, no. 5, pp. 63–68, 2005.
[17] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: Practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th Annual International Conference on Virtual Execution Environments (VEE '12)*, 2012.
[18] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014.
[19] M. A. Laurenzano, M. M. Tikir, L. Carrington, and A. Snavely, "PEBIL: Efficient static binary instrumentation for Linux," in *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS '10)*, 2010.
[20] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood, "Pin: Building customized program analysis tools with dynamic instrumentation," in *Proceedings of the 26th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '05)*, 2005.
[21] S. McCamant and G. Morrisett, "Evaluating SFI for a CISC architecture," in *Proceedings of the 15th USENIX Security Symposium (USENIX Security '06)*, 2006.
[22] J. Ming, D. Wu, G. Xiao, J. Wang, and P. Liu, "TaintPipe: Pipelined symbolic taint analysis," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, 2015.
[23] S. Nanda, W. Li, L.-C. Lam, and T.-C. Chiueh, "BIRD: binary interpretation using runtime disassembly," in *Proceedings of the 4th Annual International Symposium on Code Generation and Optimization (CGO '06)*, 2006.
[24] N. Nethercote and J. Seward, "Valgrind: A framework for heavyweight dynamic binary instrumentation," in *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, 2007.

[25] P. O'Sullivan, K. Anand, A. Kotha, M. Smithson, R. Barua, and A. Keromytis, "Retrofitting security in COTS software with binary rewriting," in *Future Challenges in Security and Privacy for Academia and Industry (IFIP SEC '11)*, 2011.

[26] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentationwith applications to taint-tracking," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO '08)*, 2008.

[27] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, 2015.

[28] 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 *Proceedings of the 10th International Conference on Information and Communications Security (ICISC '08)*, 2008.

[29] S. Wang, P. Wang, and D. Wu, "Reassembleable disassembling," in *Proceedings of the 24th USENIX Security Symposium (USENIX Security '15)*, 2015.

[30] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[31] J. Zeng, Y. Fu, K. A. Miller, Z. Lin, X. Zhang, and D. Xu, "Obfuscation resilient binary code reuse through trace-oriented programming," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*, 2013.

[32] C. Zhang, T. Wei, Z. Chen, L. Duan, L. Szekeres, S. McCamant, D. Song, and W. Zou, "Practical control flow integrity and randomization for binary executables," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.

[33] M. Zhang, R. Qiao, N. Hasabnis, and R. Sekar, "A platform for secure static binary instrumentation," in *Proceedings of the 10th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE '14)*, 2014.

[34] M. Zhang and R. Sekar, "Control flow integrity for COTS binaries," in *Proceedings of the 22nd USENIX Security Symposium (USENIX Security '13)*, 2013.

[35] Q. Zhao, I. Cutcutache, and W.-F. Wong, "PiPA: Pipelined profiling and analysis on multi-core systems," in *Proceedings of the 2008 International Symposium on Code Generation and Optimization (CGO '08)*, 2008.