SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting

Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang, *The Ohio State University*

https://www.usenix.org/conference/usenixsecurity21/presentation/chen-sanchuan

This paper is included in the Proceedings of the
30th USENIX Security Symposium.

August 11–13, 2021

978-1-939133-24-3

Open access to the Proceedings of the
30th USENIX Security Symposium
is sponsored by USENIX.

SELECTIVETAINT: Efficient Data Flow Tracking With Static Binary Rewriting

Sanchuan Chen Zhiqiang Lin Yinqian Zhang
The Ohio State University
{chen.4825, lin.3021, zhang.834}@osu.edu

Abstract

Taint analysis has been widely used in many security applications such as exploit detection, information flow tracking, malware analysis, and protocol reverse engineering. State-of-the-art taint analysis tools are usually built atop dynamic binary instrumentation, which instruments at every possible instruction, and rely on runtime information to decide whether a particular instruction involves taint or not, thereby usually having high performance overhead. This paper presents SELECTIVETAINT, an efficient selective taint analysis framework for binary executables. The key idea is to selectively instrument the instructions involving taint analysis using static binary rewriting instead of dynamic binary instrumentation. At a high level, SELECTIVETAINT statically scans taint sources of interest in the binary code, leverages value set analysis to conservatively determine whether an instruction operand needs to be tainted or not, and then selectively taints the instructions of interest. We have implemented SELECTIVETAINT and evaluated it with a set of binary programs including 16 coreutils (focusing on file I/O) and five network daemon programs (focusing on network I/O) such as `nginx` web server. Our evaluation results show that the binaries statically instrumented by SELECTIVETAINT has superior performance compared to the state-of-the-art dynamic taint analysis frameworks (e.g., 1.7x faster than that of `libdft`).

1 Introduction

One of the mostly used techniques in software security is dynamic taint analysis [28], also called dynamic data flow tracking (DDFT), which tracks the data flow of interest during program execution and has been widely used in many security applications, such as exploit detection [14, 28–30], information flow tracking [34, 41], malware analysis [18, 39], and protocol reverse engineering [10, 19]. However, the implementation of taint analysis often has high performance overhead. For instance, a state-of-the-art dynamic taint analysis framework `libdft` [17] imposes about 4x slowdown for `gzip` when compressing a file.

There has been a body of research that seeks to improve the performance of taint analysis. For instance, Jee et al. [16] applied compiler-like optimizations to eliminate redundant logic in taint analysis code. SHADOWREPLICA [15] improved the performance by decoupling taint logic from program logic, minimizing the information needed to communicate, and optimizing the shared data structures between them. TAINTPIPE [25] explored a parallel and pipeline scheme. STRAIGHTTAINT [24] combined an online execution state tracing and offline symbolic taint analysis for further performance improvement.

Interestingly, these general DDFT frameworks and their optimizations are built atop dynamic binary instrumentation (DBI), particularly Intel's PIN [22], to instrument the taint analysis logic at runtime. We believe a fundamental reason of using DBI for these frameworks is to basically avoid the code discovery challenge from static binary analysis. Note that PIN is a DBI tool, and it dynamically disassembles, compiles, and reassembles the executed code at runtime without any code discovery issues. The core module of PIN is a virtual machine (VM) that consists of a just-in-time (JIT) compiler, an emulator, and a dispatcher. PIN also has a rich set of APIs used for Pintool's implementations. However, the VM and APIs both add additional performance overhead to a taint analysis tool.

Unlike DBI, static binary instrumentation (SBI) inserts the analysis code directly into the native binary and thus avoids the unnecessary DBI overhead incurred by JIT and emulation. Meanwhile, SBI would have fewer context switches, since the rewritten binary has a better code locality. While it is challenging to perform static binary analysis, recently there are substantial advancements in static binary rewriting and reassembling (e.g., UROBOROS [37], RAMBLR [36], MULTIVERSE [6], and recently Datalog Disassembly [13]). Therefore, it is worthwhile revisiting the taint analysis and study the feasibility of using static binary rewriting for more efficient taint analysis.

In addition to the use of DBI, existing taint analysis frameworks also instrument the binary code at every possible instruction that can contribute the information flow, and rely on the execution context to determine whether there is a need to taint the corresponding operand. However, if a static analysis could figure out precisely the instructions that will never get involved in taint analysis (e.g., via some conservative static analysis), it would have not instrumented them. Therefore, enabling taint analysis to selectively instrument the binary code statically is viable and highly desired.

In this paper, we propose SELECTIVETAINT, an efficient selective taint analysis framework for binary code with static binary rewriting. There are two salient features in SELECTIVETAINT. First, it directly removes the overhead from dynamic binary translation, and is built atop SBI instead of DBI. Second, it scans taint sources of interest in the binary, statically determines whether an instruction operand will be involved in taint analysis by leveraging the value set analysis (VSA) [3,4], and then selectively taints the instructions of interest. There are well-known challenges that SELECTIVETAINT must address, such as how to deal with point-to (i.e., alias) analysis inside binary code. SELECTIVETAINT solves this problem by conservatively identifying the memory addresses that will never be involved in taint, and then taint the rest.

We have implemented SELECTIVETAINT atop SBI and evaluated it with a variety of applications consisting of 16 coreutils, and five network daemon programs such as Nginx web server. We use these programs as the benchmarks because they represent both file I/O and network I/O, the two most common input channels used by real world programs. The evaluation results show that SELECTIVETAINT is 1.7x faster than that of `libdft`, a state-of-the-art dynamic taint analysis framework. We formally prove that SELECTIVETAINT is soundy (mostly sound) [21], and also confirm the soundness of SELECTIVETAINT by using it to detect real-world exploits against the memory corruptions vulnerabilities in a variety of software including image decoder, audio normalization, and assembler.

In short, we make the following contributions:

- We propose SELECTIVETAINT, the first static-binary-rewriting based selective taint analysis framework, to substantially improve the performance overhead incurred by earlier DBI-based approaches.
- We also present a conservative tainted instruction identification approach, which statically identifies the instructions that will never involve tainted memory or registers by using VSA and then conservatively taints the rest instructions.
- We have implemented SELECTIVETAINT, and tested with 16 coreutils and five network daemons. The evaluation results show that SELECTIVETAINT has superior performance compared to the state-of-the-art taint analysis tools such as `libdft`.

2 Background

2.1 Taint Analysis

Taint analysis is a widely used program analysis technique that tracks the flow of data of interest as they propagate during the program execution [28]. It is also referred as dynamic data flow tracking (DDFT) or dynamic taint analysis (DTA), which is usually implemented using virtualization or DBI and can be performed per-process [17] or system-wide [39].

```
1  void process(int client_sock, char *buffer, int size)
2  {
3          char ch;
4          int read_size = recv(client_sock, buffer, 2048, 0);
5          if(read_size > 0)
6          {
7                  ch = buffer[0];
8                  if(ch >= 'a' && ch <= 'z')
9                          buffer[0] = ch -32;
10                 write(client_sock, buffer, read_size);
11                 memset(buffer, 0, 1024);
12         }
13 }
14
15 int server(int client_sock)
16 {
17         int i = 0;
18         char buffer[1024] = {0};
19         for(i = 0; i < 3; i++)
20         {
21                 process(client_sock, buffer, 1024);
22         }
23         return 0;
24 }
```

Figure 1: A simplified running example

Taint analysis needs taint tags, which are markings associated to registers and memory to indicate their taint status. Taint tags can have different (1) granularities to mark the taintedness for a bit, a byte, a word, or a block of data, and (2) sizes to indicate the taintedness to be a bit—tainted or not, or an integer—which input byte tainted the data. A finer granularity enhances taint analysis precision but adds performance costs, e.g., the storage cost for tag-related data structure, whereas a coarser granularity offers less precision but better performance. When a tag size is a single bit, it can be used to represent whether a corresponding register or memory location is tainted or not; when it is an integer, it can represent which part of the input (e.g., a particular byte offset) has been propagated to the tainted registers or memory locations.

A taint analysis typically consists of three components: taint sources, taint propagation, and taint sinks. In the following, we use a simplified networking program illustrated in Figure 1, as a running example, to demonstrate how a typical taint analysis works.

- **Taint sources.** Taint sources are program points or memory locations where data of interest is introduced. Typically, taint analysis is interested in user input coming from locally or remotely. For example, in Figure 1, if we are interested in the remote input, we will taint the data stored in `buffer` right after entering the system when calling `libc` function `recv` at line 4.
- **Taint propagation.** Taint tags are propagated during the program execution according to the taint propagation rules, which are specified with respect to the semantics of each instruction, e.g., the specific operands in the instruction, and also the side-effect of the instruction. For instance, for instruction `ADD src, dst`, a taint propagation rule could specify that the new tag of `dst`

is a bit-wise OR of the tags of src and dst. In Figure 1, at line 7 ch is assigned the tainted data of buffer[0] and at line 9 buffer[0] is calculated based on tainted ch, which has a data dependency, whereas at lines 8-9 whether buffer[0] is assigned or not depends on the outcome of the predicate in the if statement, which involves a tainted ch with a control dependence between buffer[0] and ch. Note that most of the DDFT works (e.g., [15–17, 24, 25]) only consider taint propagation based on data dependencies.

- **Taint sinks.** Taint sinks are specific program instructions where taint analysis checks the existence of taint tags of interest for various security applications such as detecting control flow hijacks or information flow leakage. Common taint sinks are control flow transfer instructions for detecting control flow hijack attacks, or output system calls (e.g., write, send) for detecting information leakage attack (e.g., a tainted secret leaked out). In Figure 1, line 10 could be a taint sink for information leakage detection, since it is the libc function write that writes the content starting at buffer to client_socket.

2.2 Value Set Analysis

Value set analysis (VSA) [3, 4] is a static program analysis technique. It over-approximates the set of possible values that each data object of the program could hold at each program point, and it uses a *value set* to represent the set of memory addresses and numeric value quantities.

Memory regions and abstract locations. VSA uses an abstract memory model that separates the address space into multiple disjoint areas that are referred to as *memory regions*. Memory regions in VSA consist of: a *global region* for memory locations storing uninitialized and initialized global variables, a *stack region* per function for memory locations of activation record of a procedure, and a *heap region* per heap allocation for memory locations allocated by a particular malloc-type of function call site. An *abstract location*, i.e., an *a-loc*, is a variable-like entity which spans from one statically known location to next statically known location (not including it).

Abstract addresses and value sets. An *abstract address* in VSA is represented by a pair (memory-region, offset). A set of abstract addresses can be represented using:

$$\{i|rgn_i \mapsto \{o_1^i, o_2^i, \ldots, o_{n_i}^i\}\}$$

More specifically, when there are at most one stack memory region and one heap memory region, the value set can be specified as 3-tuple [4]:

$$(global \mapsto O^g, stack \mapsto O^s, heap \mapsto O^h)$$

abbreviated as (O^g, O^s, O^h). A set of memory offsets in each memory region is represented by a *strided-interval* (SI): $s[l,u]$, where s is the stride, l and u are lower bound and upper bound. For instance, $(\{1,3,5\}, \bot, \bot)$ could be represented using SI as $(2[1,5], \bot, \bot)$.

The analysis is performed on a control-flow graph (CFG) in which each node represents an instruction (not a basic block as VSA is calculated for each instruction) and each edge represents a control flow transfer. A transfer function that characterizes the instruction semantics is associated with each edge. Note that since the address values and numeric values are interleaved in the binary, VSA tracks address values and numeric values at the same time.

2.3 Binary Instrumentation

Binary instrumentation is the process of instrumenting binary with additional analysis code added and meanwhile maintaining the original functionality. It is a widely used technique for many important security applications such as malware analysis and binary code hardening. Binary instrumentation could be either static or dynamic.

Static binary rewriting. Static binary instrumentation (SBI), also known as static binary rewriting, modifies the binary file directly. Static binary rewriting can be performed in three ways, as summarized in RAMBLR [36]: (1) trampoline-based, (2) lifting and recompiling, (3) symbolization [40] and reassembling [37]. Specifically, in trampoline-based approaches, hooks which detour the control flow to trampolines are added to the binary. In contrast, for lifting and recompiling, the binary code will be first lifted into an intermediate representation (IR), then inserted with the code of interest in the IR, and finally compiled back. The first two approaches have been known in the community for years. Recently, symbolization and reassembling approach was proposed, in which a rewriter needs to identify the locations pointed by memory references first, and then symbolize those references. The process of converting numeric references back to symbols is called symbolization. After symbolization, the rewriter could correctly relocate binary in reassembling. The first two approaches impose significant overhead and the last approach may mix code with data and may not correctly separate them.

Dynamic binary instrumentation. Dynamic binary instrumentation (DBI) recovers the code while program is executing, which can correctly separate program code from data. However, compared with static approaches, DBI has high performance overhead. There are generally two ways to implement DBI: using a trampoline, or using just-in-time (JIT) compiling. The trampoline approach replaces the instruction with a trampoline at run-time which jumps to the instrumented analysis code, and the JIT compiling approach dynamically compiles the binary on the fly.

3 Challenges and Insights

3.1 Challenges

To clearly illustrate the challenges of selective taint analysis, we still use the example code shown in Figure 1. This

program receives three messages from a client (line 19-22), capitalizes the first character in each message if needed (line 8-9), and sends the messages back to the client (line 10). It has a buffer overflow vulnerability at line 4, when receiving the input with size larger than 1,024 bytes. The taint source of our interest is the network input stored in array `buffer`, which is tainted right after the execution of `libc` function `recv`. The taint sink of our interest is the control flow transfer instruction `ret` of function `server` at line 23, assume our objective is to detect the control flow hijacks.

Performing selective binary code taint analysis using static binary rewriting is by no means trivial. Unlike DBI-based approaches where taint analysis logic is instrumented at runtime, a SBI-based approach has to analyze and rewrite the binary statically. In addition to the challenges from static binary disassembling and rewriting (they are orthogonal to the problem we aim to solve in this paper), SELECTIVETAINT has to address at least the following unique challenge—how to determine whether a disassembled instruction needs to be instrumented by taint analysis. If so, we have to also rewrite it accordingly based on the taint semantics (e.g., whether this instruction introduces a taint sources, contributes to taint propagation, or it is a taint sink).

Essentially, the problem becomes how to determine the taintedness of an instruction according to its operands including both memory addresses and registers without executing the binary. Determining the taintedness of registers is easier compared to memory addresses, since registers can be directly identified based on names whereas a memory address cannot be easily resolved. Therefore, determining the taintedness for memory addresses is much harder in SBI. More specifically, different from DTA in which a memory address has a single runtime address at each program point, static binary taint analysis can only conservatively infer the possible values for a symbolic memory address at each program point. Except for global memory addresses, symbolic addresses of stack and heap are only in relative addresses when performing the static analysis. In addition, there are also unknown inputs (from a command line, local files and keystrokes, or remote network packet) that also make the problem hard.

3.2 Insights

It is obvious that in order to address the aforementioned challenges, it requires the inference of possible values of both registers and memory cells at each program point. Fortunately, a key enabling technique in this direction is the VSA [3, 4], which seeks to compute the possible values at each symbolic memory address and register. Therefore, with VSA, we could determine whether a particular memory address or register involves taint or not, e.g., whether it is an alias to the address of our interest, or it will hold the propagations of the tainted data.

To see exactly how VSA helps our analysis, we show the value set analysis results of our running example along with its assembly code in Table 1. At the prologue of function `server`, the initial `esp` has a value set of ($\perp$, 0x0, $\perp$), since the stack pointer address for a function is unknown statically. After executing `push %ebp` at `0x8048687`, `esp` has a value set of ($\perp$, -0x4, $\perp$). The analysis continues, and computes the rest of the VSA for each register and memory operand. With the statically computed VSA, we can easily see that `ebx` at `0x80486a9` and `eax` at `0x80486c6` have the same value sets ($\perp$, -0x410, $\perp$), and thus these two registers are actually aliased. In fact, both of them refer to the address of the local variable `buffer` defined in function `server`.

To statically analyze which instructions need to be tainted, a straw-man approach is to statically maintain tainted value sets (i.e., value sets of registers and symbolic memory that need to be tainted) at each program point. In particular, this approach checks whether the value set of any of the operand of an instruction is a subset of the tainted value set, if so, this instruction is added into the tainted instruction set; meanwhile the register or symbolic memory of the corresponding operand is also added to the tainted value sets if the taint will be propagated to this operand, and the corresponding taint rule is used to taint this instruction.

However, when analyzing real-world binaries, VSA may lose its precision due to various factors such as imprecise control flow graph (CFG), sophisticated static point-to analysis (which is an undecidable problem [31]), and unknown inputs. Consequently, as illustrated in Figure 2, we may not be able to get the ideal tainted instruction set $\mathcal{I}$ for the instructions that need to be tainted, and instead the VSA identified the must-tainted instruction set $\mathcal{I}_t$ (i.e., instructions must be tainted) can have false negatives because of the imprecision mentioned, but it will not have false positives for the must-analysis (the worst case is it can be empty, if the must-analysis cannot decide anything). On the other hand, by using VSA, we can also identify must-not-tainted instruction set $\mathcal{I}_u$ that will never be involved in taint analysis. Therefore, in order not to have any false negatives (no missing of attacks) when using taint analysis, we eventually decide to taint the instructions that are not in $\mathcal{I}_u$. The worst case of our algorithm is that the identified $\mathcal{I}_u$ is $\varnothing$, which means we taint all instructions similarly to other DBI-based taint analysis. Our key objective is to confidently enlarge $\mathcal{I}_u$ as much as possible (note that we will not have false positives when being conservative).

As in our running example, in Table 1, the instructions in light gray are identified as in must-not-tainted instruction set $\mathcal{I}_u$, the instructions in dark gray are identified as in must-tainted instruction set $\mathcal{I}_t$, and all instructions not in light gray are our conservatively tainted instructions. For each instruction, a must-not-tainted value set $\mathcal{V}_u$ is maintained and if the value set of any of its operand is a subset of $\mathcal{V}_u$, this instruction is added to $\mathcal{I}_u$. For instance, for instructions at `0x804861a` and `0x804861d` before taint introduction at `0x8048620`, must-not-tainted value set $\mathcal{V}_u$ equals value set $\mathcal{S}$, which contains all possible values at this execution point (recall VSA is a

Assembly		Value Set Examples
<server>:		
8048687	push %ebp	esp:(⊥,-0x4,⊥)
8048688	mov %esp,%ebp	ebp:(⊥,-0x4,⊥)
804868a	push %edi	
804868b	push %ebx	
804868c	sub $0x420,%esp	esp:(⊥,-0x42c,⊥)
8048692	movl $0x0,-0xc(%ebp)	
8048699	lea -0x40c(%ebp),%ebx	
804869f	mov $0x0,%eax	
80486a4	mov $0x100,%edx	
80486a9	mov %ebx,%edi	ebx:(⊥,-0x410,⊥)
80486ab	mov %edx,%ecx	
80486ad	rep stos %eax,%es:(%edi)	
80486af	movl $0x0,-0xc(%ebp)	
80486b6	jmp 80486d9	
80486b8	movl $0x400,0x8(%esp)	
80486c0	lea -0x40c(%ebp),%eax	
80486c6	mov %eax,0x4(%esp)	eax:(⊥,-0x410,⊥)
80486ca	mov 0x8(%ebp),%eax	
80486cd	mov %eax,(%esp)	
80486d0	call 80485fd<process>	
80486d5	addl $0x1,-0xc(%ebp)	
80486d9	cmpl $0x2,-0xc(%ebp)	
80486dd	jle 80486b8	
80486df	mov $0x0,%eax	
80486e4	add $0x420,%esp	
80486ea	pop %ebx	
80486eb	pop %edi	
80486ec	pop %ebp	
80486ed	ret	

Assembly		Value Set Examples
<process>:		
80485fd	push %ebp	ebp:(⊥,-0x434,⊥)
80485fe	mov %esp,%ebp	ebp:(⊥,-0x434,⊥)
8048600	sub $0x28,%esp	esp: (⊥,-0x45c,⊥)
8048603	movl $0x0,0xc(%esp)	
804860b	movl $0x800,0x8(%esp)	buffer size:(0x800,⊥,⊥)
8048613	mov 0xc(%ebp),%eax	
8048616	mov %eax,0x4(%esp)	buffer addr:(⊥,-0x410,⊥)
804861a	mov 0x8(%ebp),%eax	
804861d	mov %eax,(%esp)	
8048620	call 80484f0<recv@plt>	$\mathcal{V}_u$= $\mathcal{S}$ - (⊥,[-0x410,0x3f0],⊥)
8048625	mov %eax,-0xc(%ebp)	
8048628	cmpl $0x0,-0xc(%ebp)	
804862c	jle 8048685	
804862e	mov 0xc(%ebp),%eax	
8048631	movzbl (%eax),%eax	
8048634	mov %al,-0xd(%ebp)	
8048637	cmpb $0x60,-0xd(%ebp)	
804863b	jle 8048651	
804863d	cmpb $0x7a,-0xd(%ebp)	
8048641	jg 8048651	
8048643	movzbl -0xd(%ebp),%eax	
8048647	sub $0x20,%eax	
804864a	mov %eax,%edx	
804864c	mov 0xc(%ebp),%eax	
804864f	mov %dl,(%eax)	
8048651	mov -0xc(%ebp),%eax	
8048654	mov %eax,0x8(%esp)	
8048658	mov 0xc(%ebp),%eax	inst. is tainted, as (⊥,-0x410,⊥) ⊈ $\mathcal{V}_u$
804865b	mov %eax,0x4(%esp)	
804865f	mov 0x8(%ebp),%eax	
8048662	mov %eax,(%esp)	
8048665	call 80484a0<write@plt>	
804866a	movl $0x400,0x8(%esp)	
8048672	movl $0x0,0x4(%esp)	
804867a	mov 0xc(%ebp),%eax	
804867d	mov %eax,(%esp)	
8048680	call 80484c0<memset@plt>	
8048685	leave	
8048686	ret	

Table 1: The assembly code snippets of our running example. Instructions in light gray are identified by our analysis as in must-not-tainted instruction set $\mathcal{I}_u$, and Instructions in dark gray are in must-tainted instruction set $\mathcal{I}_t$.

flow sensitive analysis). At taint source `0x8048620`, must-not-tainted value set $\mathcal{V}_u$ is updated by removing value set (⊥,[-0x410,0x3f0],⊥) from $\mathcal{V}_u$, as the tainted buffer starts at (⊥, −0x410, ⊥) with a buffer length `0x800`. At `0x8048658`, `[ebp+0xc]` has value set (⊥, -0x410, ⊥), which is not a subset of must-not-tainted value set $\mathcal{V}_u$ and thus this instruction is not added to $\mathcal{I}_u$ and will be instrumented instead. Eventually SELECTIVETAINT will conservatively taint all instructions not in $\mathcal{I}_u$, i.e., instructions not in light gray, which consists of all instructions in $\mathcal{I}_t$, i.e., instructions in dark gray, with five additional instructions in white.

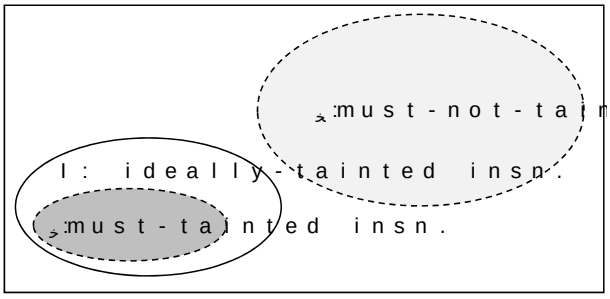


Figure 2: The Essence of SELECTIVETAINT

Scope and Assumptions. In this work, we focus on x86 binaries with ELF format running atop Linux platform. We assume the binary code is not obfuscated, and we are able to get their correct disassembly. For proof-of-concept, we demonstrate the use of taint analysis to track the untrusted user input through static binary rewriting, and detect the memory exploits by just using a single bit (tainted or not) in our taint record. Also, our static binary rewriting is based on DYNINST [7]. While it is not perfect, it has been widely used

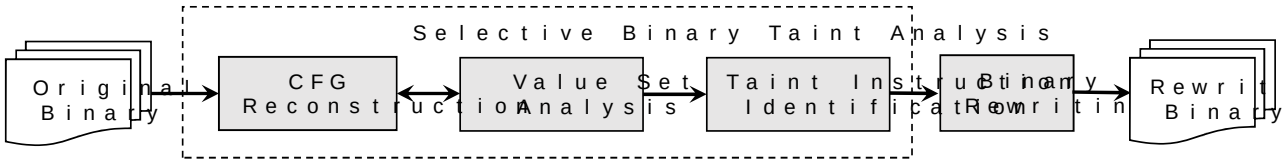


Figure 3: Overview of SELECTIVETAINT

in building many static binary rewriting-based prototypes, e.g., TYPEARMOR [35], and most recently UNTRACER [27].

4 Detailed Design

In this section, we present the detailed design of SELECTIVETAINT. As illustrated in Figure 3, there are four key components inside:

- **CFG Reconstruction (§4.1).** When given an application binary, we will first disassemble and build its CFG starting from the `main` function. If there is a library call, we will resolve the calling target and use the function summaries to decide whether further instrumentation of the library is needed. If an indirect `jmp/call` is encountered, we will perform backward slicing [36] and use the VSA and type information to resolve the target.
- **Value Set Analysis (§4.2).** VSA [3] has become a standard technique in static binary analysis for determining the possible values of a register or a symbolic memory address. We use the VSA to help identify the instruction operands that are never involved in the taint analysis.
- **Taint Instruction Identification (§4.3).** Selective tainting essentially aims to identify the instructions that are involved in the taint analysis. With the identification of $\mathcal{I}_u$ by VSA, we then start from the instructions that introduce the taint sources, and systematically identify the rest of instructions that are not in $\mathcal{I}_u$.
- **Binary Rewriting (§4.4).** Having identified the instructions that need to be tainted, we then use the static binary rewriting to insert the taint analysis logic including tracking of the taint sources and taint propagations as well as the taint checks at the taint sinks.

4.1 CFG Reconstruction

The first step of SELECTIVETAINT is to disassemble and rebuild the CFG, when given an application binary. This is quite a standard process and the only additional challenge is to identify the control flow targets of the indirect calls and jumps, as they are important to compute the VSA. To get the CFG, we first reconstruct the possible control flow targets using the RAMBLR [36] approach, and in case of undecided target (e.g., `jmp/call eax`), we use the following approaches:

Handling Indirect Call. We adopt and implement two forward-edge CFI identification approaches, namely TYPEARMOR [35] and τCFI [26], to recover the type information (i.e., parameter count and parameter type) about actual and formal parameters at the callsites and callee functions. By connecting the matching callsites and callees regarding these type information, we build a CFG which is an over-approximation of actual CFG. The type information is generated by running liveness analysis at indirect callsites and use-def analysis at callees.

Handling Indirect Jump. We first use VSA to resolve the indirect jump target and connect the jump target if it is solved. Otherwise, we determine whether the function that contains the indirect jump uses any external data references (e.g., global variable addresses): if not, we connect all of the possible basic block starting address in this function as the potential jump target (we still consider it local); otherwise, we connect the jump target with all function entry addresses. The rationale is we notice the inter-procedural jumps we encountered are from compiler optimizations, and basically compiler optimizes the call instruction with an indirect jump. We therefore connect the indirect jump in this way to get an over-approximation of the CFG.

4.2 Value Set Analysis

Our VSA Algorithm. A key technique inside SELECTIVETAINT is the VSA [3], which is a context-sensitive and flow-sensitive whole program analysis. As described in algorithm 1, our `whole_program_VSA` first initializes the `ValueSet` for each instruction in the program with an initial `esp`, initial empty heap, and initial memory cell values resolved from original binary. Then, function `VSA` is called to analyze each function `func`, which is of work list style with multiple iterations on each individual instruction until no changes are discovered (i.e., reached a fixed point). The context and value sets are adjusted depending on the type of instruction opcode, e.g., for `call/ret` instruction, inter-procedural analysis is performed and the environment is adjusted accordingly including changing the current stack region and matching formal and actual parameters.

Practical Challenges. While the idea of calculating VSA is simple, it has a number of practical challenges when used for data flow tracking, such as context-sensitive, flow-sensitivity, and alias analysis. In the following, we describe these challenges and also how we have addressed them below:

Algorithm 1: Whole Program Value Set Analysis

```
1  Function whole_program_VSA(CFG, ValueSet):
2      ValueSet, context ← init()
3      VSA(entryFunc, context)
4  Function VSA(CFG, ValueSet, func, context):
5      worklist ← {entryInst}
6      while worklist ≠ ∅ do
7          i ← pop(worklist)
8          if callInst(i) then
9              newContext ← adjustContext(context, callee(i))
10             VSA(CFG, ValueSet, callee(i), newContext)
11         if retInst(i) then
12             adjustContext(context, caller(i))
13         if condInst(i) then
14             ValueSet^i_{exit_n} ← ValueSet^i_{entry} ⊓^{VS} ValueSet_{c_n}
15         else
16             if uninitialized(op_i) then
17                 ValueSet^i_{exit}[addr(op_i)] ← (⊤,⊤,⊤)
18             newValueSet^i_{exit} ← EXE(i, ⨆_{entry_n∈entry} ValueSet^i_{entry_n})
19             if newValueSet^i_{exit} ≠ ValueSet^i_{exit} then
20                 ValueSet^i_{exit} ← ValueSet^i_{exit} ⊔ newValueSet^i_{exit}
21                 push(worklist, succs(i))
```

(I) Handling context-sensitivity. It will be overly complicated if a function is called multiple times when performing the inter-procedureal analysis in a CFG. We therefore augment our VSA with a cloning-based context sensitivity analysis [38]. Basically, we have a separate analysis for each function clone per calling context. More specifically, we generate a function clone for every acyclic path through a program call graph and, for cyclic paths, we merge all functions in a strongly connected component to have a single function context for them as in [38].

(II) Handling flow-sensitivity. Since VSA is flow-sensitive and per-instruction, it is an engineering challenge to inspect each instruction statically. We therefore borrow the idea of how symbolic execution interprets each instruction and updates the corresponding symbolic states. Essentially, when perform our flow sensitivity analysis, we need to interpret each instruction, and updates the VSA based on its semantics. Since symbolic execution is well studied (with many open source tools), we do not describe how we implement our interpreter and instead we abstract it as a simple *EXE* operation (line 18) in algorithm 1, which is responsible to capture the value set changes for each analyzed instruction (working as a transfer function in static analysis). In particular, all incoming value sets are merged on a per register and memory cell basis as input value sets and are fed into the static reasoning engine *EXE* to update the value sets of each registers and memory cells for this analyzed instruction i according to its semantics, and this updated value set forms the output of our static analysis for this particular instruction. A work list keeps looping and works on each instruction as such, until a fixed point is reached.

(III) Handling a-locs with unknown values or addresses. Performing VSA on binary suffers from the lack of dynamic information (e.g., calling context, and concrete memory addresses). One major issue when applying VSA on real-world binary is uninitialized variables and their aliases. Among these uninitialized variables, some are used in address calculation, which leads to a-locs with unknown addresses. To conservatively taint instructions, we need to infer the value set of these unknown addresses; otherwise the reads and writes to them would indicate the reads and writes to the whole address space.

In case VSA encounters an a-loc with uninitialized values or addresses due to system inputs for instance, the special handling is shown in line 16-17 in algorithm 1. In particular, our analysis will assume the uninitialized a-loc to have any value, i.e., with the value set (⊤,⊤,⊤). In practice, we have identified the following three cases in which VSA cannot determine the corresponding addresses:

(i) **Unknown values from command line input (CLI), e.g., `argv[]`.** The `argv` elements are pointers which is uninitialized at analysis-time. As shown in Figure 4a, instruction at `0x804b362` reads `argv[0]` which is unknown at analysis-time.

(ii) **Unknown addresses or values passed from missing callers.** Even we use approaches such as TypeArmor to recover CFG, there are still some callee functions without callers and the calling context is missing for these callee. As shown in Figure 4b, the function `version_etc_arn` has no identified callers, and thus, the value of parameter at instruction `0x804b7a7` is uninitialized.

(iii) **Unknown addresses or values due to library functions and system calls.** For instance, `fopen64` function returns a pointer which is a pointer to `FILE` struct that is uninitialized at analysis-time as illustrated in Figure 4c.

4.3 Taint Instruction Identification

After our whole program VSA analysis, we next need to identify the instructions that need to be instrumented for the taint analysis with the computed VSA. To this end, we have to decide whether a memory address involves taint or not, which essentially leads to problem of point-to (i.e., alias) analysis. However, due to the imprecision of the static point-to analysis, we may not be able to resolve all memory addresses with VSA [3,4], and instead we focus on identifying the addresses that will never be involved in taint analysis for each specific instruction (since VSA is flow sensitive). Initially, all instructions will be marked tainted (i.e., they will all be instrumented for taint analysis). As described in §3.2, our key objective is to minimize this set, by identifying and enlarging the must-not tainted set. In the following, we describe how we achieve this.

```
bzip2
0804b296 <main>:
 804b296: push   %ebp
 804b297: mov    %esp,%ebp
 ...
 804b2a2: mov    0xc(%ebp),%esi
 ...
 804b362: mov    (%esi),%edx
```

(a) Entry-function uninitialized variable

```
comm
0804b7a0 <version_etc_arn>:
 804b7a0: push   %ebp
 804b7a1: push   %edi
 804b7a2: push   %esi
 804b7a3: push   %ebx
 804b7a4: sub    $0x5c,%esp
 804b7a7: mov    0x74(%esp),%eax
 804b7ab: mov    0x70(%esp),%esi
 804b7af: mov    0x78(%esp),%edx
 804b7b3: mov    0x7c(%esp),%ecx
 804b7b7: test   %eax,%eax
 804b7b9: mov    0x80(%esp),%ebx
 804b7c0: mov    0x84(%esp),%edi
```

(b) Incomplete CFG caused uninitialized variable

```
cut
08049dd0 <cut_file>:
 8049dd0:push   %ebp
 8049dd1: push   %edi
 8049dd2: push   %esi
 8049dd3: push   %ebx
 8049dd4: sub    $0x4c,%esp
 ...
 8049df3: call   8048e70 <fopen64@plt>
 8049df8: test   %eax,%eax
 8049dfa: mov    %eax,%ebp
```

(c) `fopen64` uninitialized variable

Figure 4: Example code of uninitialized variable in whole program VSA

4.3.1 Must-not Tainted Analysis

In order to statically identify instructions never involved in taint analysis, we should know the must-not tainted value set, which is an opposite, more conservative counter-part of the intuitive tainted value set, at each program point. This is also a data flow analysis problem, and we have to inspect each instruction to decide whether its operand will never be involved in taint or not.

Identification Policy. Must-not-tainted set is based on the following policy: (1) instructions unreachable from taint sources are *removed* from the must-not-tainted set (which is one of the differences compared to DBI-based taint implementations), e.g., in Figure 5a, the instruction at `80491b7`, which is at the beginning of the program, is removed from must-not-tainted set as `804b4e1` is the first instruction that introduces the taint; (2) instructions with operands of potentially tainted or unknown value sets are *removed* from must-not-tainted set such as the instruction at `8055c41` that may contain tainted data from `__IO_getc` function return value in Figure 5b; (3) instructions whose operands hold literal values are added to must-not-tainted set since none of the operands will be tainted, e.g., instruction `inc %ebp` whose operand contains a literal value in register `ebp` as shown in Figure 5c is *added* to must-not-tainted set; (4) instructions whose opcode indicates they will not be involved in taint propagation are *added* to must-not-tainted set, e.g., control-flow instructions (e.g., `jmp` in Figure 5d) and compare and test instructions (e.g., `cmp`, and `test`). The must-not tainted value set will propagate along with data flow, and it is a must analysis.

```
80491b7: mov    %eax,0x8052160
...
804b4e1: call   8048d70 <read@plt>
```

(a) Unreachable instructions

```
8055c3c: call   8048f30 <_IO_getc@plt>
8055c41: mov    %eax,%edx
```

(b) Potentially tainted instructions

```
8096a07: inc    %ebp
```

(c) Untainted operand instructions

```
8062456: jmp    806238b <mbslen+0x8b>
```

(d) None taint-propagation instructions

Figure 5: Example code of the corresponding identification policy

Resolving operand's addresses. To conservatively track the must-not tainted value sets, we have to look into different types of memory access of an instruction operand: (1) for constant memory address, e.g., `[0x8000200]`, we can easily infer that it is a global variable rather than a local variable or a heap variable and the must-not tainted value sets of that address can be updated based on the constant memory address, e.g., if this constant memory address may be tainted, the constant memory address would be removed from our must-not-tainted set; (2) for a memory access based on `ESP` register, which we call *stack pointer addressing*, e.g., `[esp + 0x4]`, we identify it as a stack variable, the stack region and offset can be obtained through our whole program analysis caller/callee stack information; (3) for a memory access without `ESP` register, e.g., `[eax]`, this is tricky since we may not know whether it is a stack, global or heap variable; we thus use the VSA result to decide the value set of the memory access: if the VSA cannot decide whether the memory access address is tainted or not, we conservatively remove it from the must-not tainted set.

Resolving operand's values. Once the algorithm meets an instruction operand that is uninitialized (it can lead to an

alias that cannot be resolved), as mentioned in §4.2, we conservatively taint the associated variables, depending on the specific cases:

(i) **Unknown value from CLI (e.g., Figure 4a).** Based on where the input value is going to be stored, we assign a corresponding uninitialized value for these variables. For instance, we will assign an uninitialized value for a stack varaible which belongs to the stack of the caller of `main` and is prior to the stack of `main` function, and proceed the must-not analysis as usual.

(ii) **Unknown value passed from missing callers (Figure 4b).** A caller function passes function parameters to a callee function, causing aliasing between actual parameters and formal parameters. When CFG reconstruction cannot determine the callers for a callee, it results in unknown value from the missing callers. We conservatively remove all of the memory access instructions in the function and all of the data uses of these variables outside the function from the must-not tainted set. To optimize our analysis, we do not taint all global variables, and instead we taint the data based on their types (the type inference is described below) in global sections such as `.data` and `.bss`.

(iii) **Unknown value due to library function calls and system calls (Figure 4c).** We taint these unknown variables according to the semantics of library functions and system calls. For instance, the pointer returned by `fopen` is put in the must-not tainted set at the program point right after the library call and the pointer returned is assigned a value set in a special heap region.

Variable type inference. To taint instructions more precisely, we perform a simple variable type inference to determine whether a variable is a pointer or not. We care them because we want to identify the potential pointers that can hold the tainted buffer. The analysis is based on whether a variable is dereferenced or whether it is a pointer type parameter or return value of known library functions as type sinks [20]. For instance, `movzbl (%ebx),%eax` indicates the variable stored at `ebx` is a pointer, and also variable stored at `edi` in the following snippet is a pointer as it is passed to the first parameter of `strchr` library function. With variable type inference, we could only taint poniter variable of interest when an unknown pointer is dereferenced instead of tainting all variables.

```
movl $0xa,0x4(%esp)
mov %edi,(%esp)
call <strchr@plt>
```

Our Algorithm. Specifically, the must-not tainted analysis algorithm as shown in algorithm 2 first scans the whole binary for possible taint sources, e.g., `read` system call and `recv` system call (line 2). Each identified taint source serves as a starting point of our analysis. The initial tainted buffer has two major characteristics: start address and length. As

Algorithm 2: Must-not Tainted Analysis

```
1  Function MustNotTainted(UntaintedSet, TaintedInst, ValueSet):
       input  : set of must-not tainted data object UntaintedSet, set of
                tainted instructions TaintedInst, value set ValueSet
       output : set of tainted data object UntaintedSet, set of tainted
                instructions TaintedInst
2      Source ← TaintSourceScan(Bin)
3      Init(buffer_start_addr, buffer_length, ValueSet, Source)
4      if unbounded(ValueSet^i_entry[buffer_startaddr]) ∨
         unbounded(ValueSet^i_entry[buffer_length]) then
5          exit()
6      while changed do
7          foreach instruction i do
8              if ValueSet^i_entry[opaddr] ∉ UntaintedSet then
9                  TaintedInst ← TaintedInst ⊔ {i}
10                 Transfer(UntaintedSet, ValueSet)
11 Function Transfer(UntaintedSet, ValueSet, i):
12     switch rule(i) do
13         case tag(opaddr_dest) ← tag(opaddr_dest) | tag(opaddr_src) do
14         case tag(opaddr_dest) ← tag(opaddr_src) do
15         case tag(opaddr_unary) ← tag(opaddr_unary) do
16             UntaintedSet ← UntaintedSet −
                 ValueSet^i_entry[opaddr_dest]
17             if ValueSet^i_entry[opaddr_src] ⊆ UntaintedSet ∧
                 evalToConcrete(ValueSet^i_entry[opaddr_dest]) then
18                 UntaintedSet ← UntaintedSet ⊔
                     ValueSet^i_entry[opaddr_dest]
19             else if ValueSet^i_entry[opaddr] = (⊤,⊤,⊤) then
20                 UntaintedSet ← UntaintedSet −
                     ValueSet^i_entry[Overtaint(opaddr_dest)]
```

our evaluation shows, we are able to identify the value set of the starting address and length of the buffer that introduces the taint. Otherwise, if either upper bound or lower bound of buffer's starting address or buffer length cannot be determined, our analysis triggers a warning and terminates, since it may indicate program vulnerability (line 4-5). The analysis is of a work-list style and iterates over each instruction until the `UntaintedSet` and `TaintedInst` remain unchanged (reached a fixed point). For each instruction i in the program (line 7-10), we first compare the incoming value sets of instruction operand address with our must-not untainted value sets. If the former is not a subset of the latter, the instruction is identified as a possible tainted instruction for the later taint propagation logic instrumentation (line 9). We then process `UntaintedSet` with respect to the taint propagation rule of each instruction (line 10). Particularly, if the taint propagation rule for instruction i decides that i has a data flow dependence between instruction operand(s), i.e., the taint propagation rule is in the form of:

$$\begin{aligned} \text{tag}(opaddr_{dest}) &\leftarrow \text{tag}(opaddr_{dest}) \mid \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{dest}) &\leftarrow \text{tag}(opaddr_{src}) \\ \text{tag}(opaddr_{unary}) &\leftarrow \text{tag}(opaddr_{unary}) \end{aligned}$$

we taint the destination operand and remove it from `UntaintedSet` as shown in line 16. If the source operand

is deemed untainted and we know the exact concrete address of the destination operand, we enlarge our `UntaintedSet` by adding destination operand value set to `UntaintedSet` as illustrated in line 17-18. Otherwise, we conservatively taint all of the possible memory address involved in instruction i.

Example. We use the instruction `0x8048634: mov %al,-0xd(%ebp)` listed in Table 1 to demonstrate how to use our propagation rules to update the must-not tainted set. Specifically, since the source operand of instruction `0x8048634` is not in `UntaintedSet`, this instruction is added to `TaintedInst` for further taint propagation logic instrumentation and since the taint propagation rule for this instructions is in the form of:

$$\text{tag}(op_{dest}) \leftarrow \text{tag}(op_{dest}) \mid \text{tag}(op_{src})$$

According to algorithm 2, `UntaintedSet` is updated:

$$\begin{aligned} \text{UntaintedSet} &= \text{UntaintedSet} - \text{ValueSet}^{i}_{entry}[opaddr_{dest}] \\ &= \text{UntaintedSet} - (\perp, \text{-0x440}, \perp) \end{aligned}$$

4.3.2 Soundness Analysis of SELECTIVETAINT

We define *false negatives* to be the instructions that SELECTIVETAINT considers to not be tainted but the ground truth indicates it should, and *false positives* to be the instructions that SELECTIVETAINT considers to be tainted but the ground truth indicates it should not. Therefore, we can afford to have false positives, since the overly tainted instructions will only impact the performance overhead (we could have avoided tainting them). In contrast, we should avoid having false negatives; otherwise, we could miss the attacks if we do not taint these instructions.

In the following, we provide a formal analysis of our must-not tainted analysis to prove that it is *soundy* [21] (mostly sound), i.e., it hardly introduces false negatives, which means all instructions in the must-not-tainted instruction set $\mathcal{I}_u$ generated by our must-not tainted analysis are indeed not-tainted. It is soundy, as there could be the imprecise CFG and VSA in practice.

Figure 6 shows the formal representation of must-not tainted analysis. Basically, for removing or adding an instruction i in $\mathcal{I}_u$, one of the four primary inference rules, i.e., rule UNREACHABLE, UNKNOWNOPERAND, UNTAINTEDOPERAND, or NONPROPAGATEOPCODE has to be applied:

- In UNREACHABLE rule, if there is no path from taint sources to instruction i and no path from instruction i to taint sources, then i is removed from the must-not-tainted instruction set $\mathcal{I}_u$;
- In UNKNOWNOPERAND rule, if there exists an operand with unknown value set, then i is removed from the must-not-tainted instruction set $\mathcal{I}_u$;
- In UNTAINTEDOPERAND rule, if for each operand o of instruction i, its value set is a subset of must-not-tainted value set $\mathcal{V}_u$, then i is added to the must-not-tainted instruction set $\mathcal{I}_u$;
- In NONPROPAGATEOPCODE rule, if for each operand o of instruction i, there is no side effect on operand o after executing i, then i is added to the must-not-tainted instruction set $\mathcal{I}_u$.

The rest rules of Figure 6 are auxiliary inference rules. Rule REACHABLE indicates, if instruction i_2 is a successor of instruction i_1, then i_2 is reachable from i_1. Similarly, TRANSREACHABLE rule indicates, if instruction i_3 is a successor of instruction i_2 and instruction i_2 is a successor of instruction i_1, i_3 is reachable from i_1. Note the succ predicate in TRANSREACHABLE rule includes the control flow transfer for fall-throughs, unconditional jumps, conditional jumps, calls, returns, and other control flow transfer which is represented as an edge in the whole program control flow graph. LITERALOPERAND rule indicates, if the operand value set has a type of `literal`, it is added to must-not-tainted value set $\mathcal{V}_u$ and LABELOPERAND rule indicates, if the operand's value set has a type of `label`, it is added to must-not-tainted value set $\mathcal{V}_u$. TAINTSOURCE and TAINTPROPAGATE rules infer that the tainted value set is removed from must-not-tainted value set $\mathcal{V}_u$ and when taint propagates from source operand to destination operand of an instruction, the destination operand is also tainted. PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE rules indicate that, if after executing instruction i, the value sets of instruction operands are not changed and only the value sets of program counter or status registers are changed, then instruction i is added to must-not-tainted instruction set $\mathcal{I}_u$.

Theorem 1. *Must-not-tainted analysis is sound, except for the precision loss due to imprecise CFG and VSA results (thereby making it soundy or mostly sound).*

Proof. We prove this theorem with *induction*.

(1) In the first iteration of the analysis, the must-not-tainted set $\mathcal{I}_u$ is $\varnothing$. Must-not-tainted analysis is sound since every instruction is tainted.

(2) We next prove that if in the k^{th} iteration, the must-not-tainted analysis is sound, w.r.t, precise CFG and VSA results, it also holds in $(k+1)^{th}$ iteration.

Suppose the must-not-tainted set $\mathcal{I}_u$ in the k^{th} and $(k+1)^{th}$ iteration are $\mathcal{I}_u^k$ and $\mathcal{I}_u^{k+1}$, respectively. Given any instruction i, whether instruction i is added to or removed from the must-not-tainted instruction set $\mathcal{I}_u^k$ has four cases:

(2.1) *Instruction i cannot be reached from taint sources and taint sources cannot be reached from instruction i.* The UNREACHABLE rule will remove i from $\mathcal{I}_u^k$. In this case, no paths lies between instruction i and taint sources, which indicates incomplete CFG and thus instruction i can be potentially tainted, and therefore safely removing i from $\mathcal{I}_u^k$ will result in a sound $\mathcal{I}_u^{k+1}$.

CFG imprecision. As reconstructing CFG is a hard problem in practice, case 2.1 is sound except for the imprecise CFG.

Primary Inference Rules

Instructions:

$$\textsc{Unreachable}\ \frac{\nexists i_s \in source,\ i_s \rightsquigarrow i,\ i \rightsquigarrow i_s}{\mathcal{I}_u\ -{=}\ \{i\}} \qquad \textsc{UnknownOperand}\ \frac{\exists o \in op(i),\ V[o] = (\bot,\bot,\bot)}{\mathcal{I}_u\ -{=}\ \{i\}}$$

$$\textsc{UntaintedOperand}\ \frac{\forall o \in op(i),\ V[o] \subseteq \mathcal{V}_u}{\mathcal{I}_u\ \cup{=}\ \{i\}} \qquad \textsc{NonPropagateOpcode}\ \frac{\forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

Auxiliary Inference Rules

Control-flows:

$$\textsc{Reachable}\ \frac{succ(i_1,\ i_2)}{i_1 \rightsquigarrow i_2} \qquad \textsc{TransReachable}\ \frac{succ(i_1,\ i_2) \quad succ(i_2,\ i_3)}{i_1 \rightsquigarrow i_3}$$

Operands:

$$\textsc{LiteralOperand}\ \frac{l \in op(i) \quad l : \texttt{literal}}{\mathcal{V}_u\ \cup{=}\ V[l]} \qquad \textsc{LabelOperand}\ \frac{l \in op(i) \quad l : \texttt{label}}{\mathcal{V}_u\ \cup{=}\ V[l]}$$

$$\textsc{TaintSource}\ \frac{o \in taintedop(i_s) \quad i_s \in source}{\mathcal{V}_u\ -{=}\ V[o]} \qquad \textsc{TaintPropagate}\ \frac{o_1 \in sourceop(i) \quad o_2 \in destop(i) \quad V[o_1] \subseteq \mathcal{V}_u}{\mathcal{V}_u\ -{=}\ V[o_2]}$$

Opcodes:

$$\textsc{PCRegChangeOpcode}\ \frac{V[pc] \overset{i}{\not\equiv} V[pc] \quad \forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

$$\textsc{StatusRegChangeOpcode}\ \frac{V[status] \overset{i}{\not\equiv} V[status] \quad \forall o \in op(i),\ V[o] \overset{i}{\equiv} V[o]}{\mathcal{I}_u\ \cup{=}\ \{i\}}$$

Figure 6: The formal inference rules of our must-not tainted analysis. $\mathcal{V}_u$: Must-not-tainted value set, $\mathcal{I}_u$: Must-not-tainted instructions, $\overset{i}{\equiv}$: Equal values after executing i, V: VSA result map.

As shown in §4.1, SELECTIVETAINT matches callers and callees based on forward-edge CFI identification approaches and matches jumps and jump targets with basic block starting addresses within the same function or function entry addresses. These methods may introduce false negatives and produce imprecise CFG for real-world binaries.

(2.2) *One or more operands of instruction i have an unknown value set.* The UNKNOWNOPERAND rule will remove i from $\mathcal{I}_u^k$. In this case, instruction i can propagate taints, and therefore safely removing i from $\mathcal{I}_u^k$ will result in a sound $\mathcal{I}_u^{k+1}$.

VSA imprecision. Though VSA may introduce imprecision, this rule conservatively removes all instructions with unknown value sets from $\mathcal{I}_u^k$.

(2.3) *All operands of instruction i are subsets of must-not-tainted value set* $\mathcal{V}_u^k$. $\mathcal{V}_u^k$ is updated based on rules in the Operands rule group in Figure 6. Per rule LITERALOPERAND and LABELOPERAND, if an operand is of type `literal` or `label`, its value cannot propagate taint and it is added to $\mathcal{V}_u^k$. Per rule TAINTSOURCE and TAINTPROP, at taint source and taint propagation instructions, $\mathcal{V}_u^k$ gets updated by removing the tainted value set. If all operands of instruction i are subsets of must-not-tainted value set $\mathcal{V}_u^k$, it means all values in the operands are must-not-tainted, and it is sound after adding it to $\mathcal{I}_u^{k+1}$ from $\mathcal{I}_u^k$.

VSA imprecision. As VSA is an undecidable problem, it may introduce imprecision when VSA fails to identify whether an operand is actually a subset of $\mathcal{V}_u^k$ and when $\mathcal{V}_u^k$ is updated. Thus, in practice, this rule leads to a soundy analysis, i.e., mostly sound except for the imprecision caused by imprecise VSA.

(2.4) *Instruction opcode has no impact on taint propagation.* In this case, instruction i should be added to $\mathcal{I}_u$, as the instruction does not involve in any explicit handling of tainted data. Particularly, an instruction may only have side effects on program counter or status register but not its operands, and in these cases no taint is involved and instruction i should be added to $\mathcal{I}_u^k$, as in rules PCREGCHANGEOPCODE and STATUSREGCHANGEOPCODE, which results in a sound $\mathcal{I}_u^{k+1}$.

Therefore, must-not-tainted analysis of SELECTIVETAINT is soundy, due to the imprecision introduced by current limitations of undecidable CFG reconstruction and VSA results in binary (otherwise, it is sound). □

4.4 Binary Rewriting

Based on the identified tainted instructions, we instrument taint propagation logic via binary rewriting for each instruction just as how a conventional DFT performs. The only

Algorithm 3: SELECTIVETAINT Algorithm

```
1 Function SelectiveTaint(Bin):
      input  : original bianry Bin
      output : instrumented binary NewBin
2     Init (UntaintedSet, TaintedInsn, ValueSet)
3     while changed do
4         CFG ← CfgReconstructtion(Bin, CFG, ValueSet)
5         ValueSet ← whole_program_VSA(CFG, ValueSet)
6         UntaintedSet, TaintedInsn ← MustNotTainted(UntaintedSet,
            TaintedInst, ValueSet)
7     NewBin ← Rewriting (Bin, TaintedInst)
```

difference is that conventional DFTs instrument at run-time through dynamic binary instrumentation, whereas we instrument the binary statically to track how taints are introduced at the taint sources, propagated, and checked at taint sinks.

With the support from our CFG reconstruction, value set analysis, and taint instruction identification, we then sequentially combine these three analyses in a loop body and the set of `ValueSet`, `UntaintedSet` and `TaintInst` get gradually changed until a fixed point is reached. In particular, as illustrated in algorithm 3, at line 2, we first initialize the three sets of `ValueSet`, `UntaintedSet` and `TaintInst` with the taint source information. We will reach a fixed point after iterations of sequentially applying CFG reconstruction, value set analysis and taint instruction selection algorithm (line 3-6). When all the taint sources are processed, at line 7, our binary rewriter rewrites the original binary with taint propagation logic on selected instructions.

Note that for performance reasons, we use a function summary approach to process standard libraries such as `libc`, which is inspired by how RAMBLR [36] handled libraries. That is, we will not statically rewrite the instructions in the library, and instead we rewrite the callers to perform direct taint tracking, e.g., introducing taint, and propagating taint according to the corresponding parameters. For instance, when we notice `memcpy` call, we will directly taint the destination memory based on the data in the source memory.

5 Implementation

We have implemented an open source version of SELECTIVETAINT atop `angr` [33] and `Dyninst` [7]. Specifically, (1) we used `angr` to build a CFG for the binary, then implemented our own forward-edge control flow target identification based on TYPEARMOR [35] and τCFI [26], i.e., using our own VSA to determine unsolved call sites targets, connecting unsolved call sites to functions with the same parameter count and parameter type, and connecting unsolved indirect jumps with basic block starting address or all function entry addresses; (2) we implemented our own flow-sensitive and context-sensitive whole program VSA, which is used to determine the value set held at each program point; (3) based on the generated CFG and VSA results, we implemented taint instruction identification using the rules described in Figure 6 to identify the untainted instructions; and (4) after that we use `Dyninst`, which is widely used in recent studies [27, 35], to statically rewrite the binary. The total implementation of SELECTIVETAINT consists of 7,000 python code, and 22,000 C/C++ code. SELECTIVETAINT is tested in Ubuntu 14.04 32 bit OS to be compatible with the legacy 32 bit version `libdft`.

CFG Reconstruction. To implement the analyzer, first, we recover the control flow graph (CFG) of the binary using `angr`. Basically, we use `angr` to find every basic block address, its containing instructions and the predecessors and successors of each basic block. Afterwards, the remaining unsolved indirect control flow transfer targets are further resolved using our method described in §4.1.

Value Set Analysis. We first initialize the corresponding `ValueSet` with the data extracted from original binary, e.g., section and segment information, initial data values in `.rodata` and `.data` sections. With respect to the variables in different memory region: for (1) stack variable, we track the value set of stack pointer `SP` in different calling context, examine and identify whether a variable is a stack variable and in which function the variable is defined, i.e., in whose stack frame the variable resides; for (2) global variable, we track the value set of variables and check if it could be evaluated to an address in code segment or data segment as a global variable; for (3) heap variable, we track the call instructions for `malloc`-family library calls to determine whether it is a heap variable.

In intra-procedural analysis, the value set for each a-loc is calculated in a worklist algorithm until a fixed point is reached. In inter-procedural analysis, a function summary is generated based on intra-procedural analysis results to summarize the value set changes of each function. The static analysis finishes when all value sets in the whole program remain unchanged.

Taint Instruction Selection. We examine and maintain a must-not tainted a-loc set for each program point based on the value sets generated by VSA and the type of each instruction generated by CAPSTONE [2] disassembler. When must-not tainted a-loc sets reach a fixed-point, each instruction is examined as tainted or untainted based rules in Figure 6, i.e., we conservatively assume an a-loc is tainted whenever we cannot determine its taintedness. Unlike `libdft` which is implemented using PIN [22], we do not go into the dynamic library functions, and instead, we use a function summary for each library functions to track taint propagation.

Binary Rewriting. Our rewriter is implemented with a bit tag size and a byte tag granularity using `Dyninst` [7] binary instrumentation and analysis framework. `Dyninst` is an anywhere, anytime binary instrumentation framework which could be used in both static binary rewriting at compile-time or dynamic instrumentation at run-time. We favor `Dyninst` as it is a the-state-of-the-art tool in binary rewriting which is used in a variety of tools and its robust API implementation.

Benchmark	Input Functions	# Func.	# Inst.	# SELECTIVETAINT Instrum. Inst. (%)	# libdft Executed Tainted Inst. (%)	Analysis Time (s)
Utilities						
tar	read, fscanf	967	65,795	45,630 (69.35%)	4,083 (6.21%)	688
gzip	read, _IO_getc	220	15,173	10,076 (66.41%)	2,067 (13.62%)	40
bzip2	fread, fgetc	127	16,195	11,160 (68.91%)	3,524 (2.18%)	51
scp	read	145	6,390	4,238 (66.32%)	1,875(29.34%)	10
cat	read, fscanf	174	8,003	5,366 (67.05%)	548 (6.85%)	19
comm	fscanf	150	4,659	3,254 (69.84%)	918 (19.70%)	9
cut	fgetc, fscanf, __fread_chk	181	6,587	4,343 (65.93%)	742 (11.26%)	13
grep	read, fscanf, fread_unlocked	410	28,858	19,500 (67.57%)	3,693 (12.80%)	129
head	read, fscanf	149	6,533	4,517 (69.14%)	497 (7.61%)	13
nl	fscanf	252	20,677	14,082 (68.10%)	684 (3.31%)	77
od	fgetc, fscanf, fread_unlocked, __fread_unlocked_chk	188	10,696	7,143 (66.78%)	1,640 (15.33%)	24
ptx	fread, fscanf	297	25,906	17,503 (67.56%)	5,478 (21.15%)	106
shred	fscanf, __read_chk, fread_unlocked	198	9,195	6,404 (69.65%)	1,673 (18.19%)	21
tail	read, fscanf	216	10,966	7,251 (66.12%)	825 (7.52%)	26
truncate	fscanf	168	8,952	6,006 (67.09%)	491 (5.48%)	22
uniq	fscanf	165	5,539	3,822 (69.00%)	815 (14.71%)	12
Network daemons						
exim	fgetc, fread, fscanf, _IO_getc, recv	876	140,847	93,058 (66.07%)	8,160 (5.79%)	2,843
memcached	read, fgets, recvfrom	286	19,319	13,676 (70.79%)	852 (4.41%)	62
proftpd	read, fgets, __read_chk	1,037	153,306	106,821 (69.68%)	19,181 (12.51%)	3,048
lighttpd	read, fread	466	31,130	21,713 (69.75%)	5,437 (17.47%)	156
nginx server	read, pread64, readv, recv	1,277	133,666	92,041 (68.86%)	12,905 (9.65%)	2,249
Other applications						
SoX 14.4.2	read,fread,fgets,_IO_getc,__isoc99_scanf,__isoc99_fscanf	1,159	112,762	67,808 (60.13%)	9,583 (8.50%)	1,791
TinTin++ 2.01.6	read,fread,fgets,_IO_getc,gnutls_record_recv,fgetc	831	93,618	74,389 (79.46%)	9,839 (10.51%)	975
dcraw 9.28	fread,fscanf,__fread_chk,_IO_getc,fgets,jpeg_read_header	292	70,358	42,840 (60.89%)	967 (1.37%)	510
ngiflib 0.4	fread,_IO_getc	43	2,135	1,477 (69.18%)	649 (30.40%)	3
Gravity 0.3.5	read,getline	1,124	86,783	65,636 (75.63%)	16,959 (19.54%)	773
MP3Gain 1.5.2	fread,_IO_getc	144	17,573	9,900 (56.34%)	5,934 (33.77%)	53
NASM 2.14.02	fread,fgets,fgetc	826	87,456	66,601 (76.15%)	7,276 (8.32%)	838
Jhead 3.00	fread,fgetc	118	9,815c	6,805 (69.33%)	621 (6.33%)	19

Table 2: Statistics of the instrumented instructions by SELECTIVETAINT and `libdft`

6 Evaluation

In this section, we present the evaluation results. To see the improvements over dynamic taint analysis, we compare SELECTIVETAINT with `libdft` [17]. The version of Intel Pin used to build `libdft` was 2.14 (build 71313), and we slightly modified the `nullpin` and `libdft` tool and adopted them in our experiment settings. Also, to see the advancements of the selective taint analysis, we also implemented a static taint analysis by instrumenting all instructions, and we call this system STATICTAINTALL. We use four commnly used Unix utilities, i.e., the GNU versions of `tar` (version 1.27.1), `gzip` (version 1.3.13), `bzip2` (version 1.0.3), `scp` from OpenSSH (version 3.8), 12 file content processing utilities `cat`, `comm`, `cut`, `grep`, `head`, `nl`, `od`, `ptx`, `shred`, `tail`, `truncate`, `uniq` from `coreutils` (version 8.21) and `grep` (version 2.16), and we also use email server `exim` (version 4.80), general-purpose distributed memory caching system `Memcached` (version 1.4.20), FTP server `ProFTPD` (version 1.3.5), web server `lighttpd` (version 1.4.35) and `nginx` (version 1.4.0), and eight recent programs (each of which contains a memory corruption vulnerability) to evaluate SELECTIVETAINT. We first evaluate its effectiveness by looking into the details of how SELECTIVETAINT performs in §6.1, and then report the performance overhead of the rewritten binaries in §6.2. Finally, we demonstrate its security applications with real world binaries in §6.3.

6.1 Effectiveness

We report the effectiveness of how SELECTIVETAINT performs with the common Unix utilities `tar`, `gzip`, `bzip2`, `scp`, `cat`, `comm`, `cut`, `grep`, `head`, `nl`, `od`, `ptx`, `shred`, `tail`, `truncate`, `uniq`, network daemons `exim`, `memcached`, `proftpd`, `lighttpd`, `nginx`, and eight other applications in Table 2. The first column shows the 29 C/C++ programs in the benchmark we used in our evaluation, followed by the 2nd column of the input functions detected by SELECTIVETAINT. Note that the input function is the function that introduces the taint sources. Next, we report the total number of functions contained in the benchmark program in the 3rd column, which provides an estimation of the complexity of the program. Then, we show the total number of instructions identified in the binary in the 4th column. Our STATICTAINTALL statically rewrites all of these instructions, similarly to how dynamic taint analysis instruments them. This will provide an upper bound of how SELECTIVETAINT would perform in the worst case (by statically taint them all). Next, we show the total number of instructions that need to be statically

instrumented by SELECTIVETAINT in the 5th column followed by the total number of executed unique instructions that really involved in taint analysis in the 6th column, and this number is obtained by running the corresponding benchmark by using the default configured input with `libdft`, which will provide a lower bound of the number of unique tainted instructions. For fair comparison, we did not count the instructions in the library from the `libdft` trace since SELECTIVETAINT will not instrument them. Finally, we report how long SELECTIVETAINT performs to process each of the benchmarks in the last column.

We can observe from Table 2 that our static analysis works well in these benchmarks, and we have reduced the possible tainted instructions to about 56.34% - 79.46% compared to STATICTAINTALL. While ideally we would like to instrument only the instruction involved in the taint analysis (which is a subset of the instructions identified by SELECTIVETAINT), as detected by the `libdft` which shows about 1.37% - 33.77% of these instructions are essentially needed in the taint analysis at run-time with an average of 6.85% instructions, we will not be able to achieve this by purely static analysis.

False Positives and False Negatives. We have defined *false positives* and *false negatives* in Section 4.3.2. By examining the instructions tainted by SELECTIVETAINT and `libdft`, we observe SELECTIVETAINT reports no false negative but false positives. False positives indicate SELECTIVETAINT is conservative and has over-tainted instructions, and such false positives are acceptable (it will not miss any attacks). Meanwhile, no false negative indicates our approach is a sound over-approximation of the tainted instructions. This is attributed to the conservative rules in Figure 6; for instance, we remove the value set from untainted value set whenever we cannot determine the taintedness of that value set. Note the 6th column in Table 2 is generated by running the tested benchmarks, which may not explore all instructions that should be tainted and thus the ground truth is unavailable and we only observe false positives without quantifying them.

Internal Statistics. We also measured the statistics of SELECTIVETAINT in Table 3 to understand its inner-workings. Columns 2-3 are CFG construction details, i.e., the number of initial CFG edges and the number of final CFG edges after our CFG construction. We can observe our CFG construction can add hundreds of edges to the CFG using the techniques described in §4.1. Columns 4-8 are value set analysis statistics, which are the number of a-locs in the analysis, the unknown a-locs due to command line parameters, argument aliasing when missing callers, and library function calls. We can observe our approach identifies a large number of unknown a-locs in each category. Columns 9-12 are numbers in taint instruction identification, such as the number of initially untainted value sets in the first iteration, the number of final untainted value sets, the intra-procedural iteration times, and the inter-procedural iteration times. We can observe that the number of untainted value sets get smaller through analysis iterations, which means our analysis does propagate untaintedness and remove potentially tainted value sets from the must-not-tainted set $\mathcal{V}_u$. The intra-procedural analysis generally has hundreds of iterations while the inter-procedural analysis has significantly fewer iterations, from which we can observe the intra-procedural analysis reaches the fixed point with more iterations than inter-procedural analysis.

Performance. With respect to the performance (e.g., the analysis time) of SELECTIVETAINT itself, we notice it sometimes consumes tens of minutes to finish analyzing a program. This is understandable, since SELECTIVETAINT will inspect each instruction and calcaulate VSA for each of them. Meanwhile, the analysis has to be run twice: first calculating the VSA, and then determining the taintedness. We notice it took more than 50 minutes to process the `proftpd` FTP server, whereas for small binaries, e.g., `scp`, it could take just a few seconds.

6.2 Efficiency

Next, we measure the performance overhead of the rewritten binaries. To compare with `libdft`, we run the binaries with the default configured input, with nullpin (a simple implementation to evaluate Intel PIN platform overhead), and `libdft` with a bit level taint. We run the corresponding benchmark with and without rewriting to understand the additional overhead. All of the experimental results were obtained with 10 runs and then normalized by dividing each average result against native unmodified executables.

Unix Utilities. Figure 7a shows the normalized runtime overhead of nullpin, `libdft`, STATICTAINTALL, SELECTIVETAINT, when running with 16 Unix Utilities, compared with the native execution. We can notice that `libdft` imposes a slowdown ranging from 1.39 (`tar`) to 5.86x (`scp`), whereas STATICTAINTALL and SELECTIVETAINT impose 1.10x (`tar`) to 3.85x (`bzip2`) and 1.02x (`tar`) to 3.41x (`grep`), respectively. STATICTAINTALL outperforms `libdft` in all benchmarks with 1.26x - 1.87x faster with an average of 1.53x and similarly SELECTIVETAINT performs even 1.36x - 2.41x faster than that of `libdft` with an average of 1.77x.

Network Daemons. One ideal use case for SELECTIVETAINT would be for the protection of network daemons. We thus use `exim`, `memcached`, `proftpd`, `lighttpd`, `nginx` as benchmarks to thoroughly evaluate its overhead. In particular, we tested nullpin, `libdft`, STATICTAINTALL, and SELECTIVETAINT on these five daemons, in which `exim` is tested by sending email messages, `memcached` by getting values with keys from database, and ftp and http daemons via requesting files from the daemons. All four tools including `libdft` performs no more than 4x slowdown. The biggest slowdown for `libdft` is 3.76x (`exim`). STATICTAINTALL imposes 1.25x-2.23x slowdown and outperforms `libdft` by

Benchmark	CFG Reconstruction		Value Set Analysis					Taint Instruction Identification			
	Init. Edges	Updated Edges	A-Loc	Uninit. CLI	Uninit. Arg. Alias	Uninit. Lib	Uninit. Total	1st-pass Untaint-V	Last-pass Untaint-V	#Intra. Iteration	#Inter. Iteration
Utilities											
tar	34,893	64,479	18,428	6	243	2,217	2,376	11,237	8,441	6,216	5
gzip	7,389	7,792	3,506	6	97	253	356	3,272	1,892	1,345	5
bzip2	6,465	6,825	2,182	6	43	283	332	2,005	1,387	887	6
scp	3,315	3,501	1,837	7	67	246	320	1,834	1,834	289	2
cat	4,069	4,242	2,195	6	153	388	547	1,828	1,361	1,168	6
comm	2,163	2,321	1,630	6	126	199	331	1,408	1,086	1,203	7
cut	3,109	3,796	1,940	6	80	203	289	1,497	1,338	1,071	5
grep	14,730	17,444	6,709	7	101	951	1,059	5,451	4,196	2,460	5
head	3,014	3,141	1,795	7	118	252	377	1,602	1,085	1,023	6
nl	10,256	10,711	4,453	6	191	421	618	3,608	3,038	1,776	6
od	5,003	5,274	2,577	6	130	342	478	2,122	1,768	1,099	5
ptx	13,264	14,187	5,553	7	197	706	910	4,565	3,593	1,789	5
shred	4,103	4,484	2,340	6	102	328	436	1,881	1,233	1,188	5
tail	5,457	6,404	2,862	6	72	394	472	2,102	1,772	1,271	5
truncate	4,418	4,582	2,263	6	145	389	540	1,907	1,338	1,142	6
uniq	2,584	2,854	1,862	6	114	217	337	1,521	1,280	1,325	7
Network daemons											
exim	69,233	80,395	36,011	7	214	3,311	3,532	16,598	7,181	6,521	6
memcached	9,774	11,336	5,435	7	38	866	911	3,865	1,772	1,380	4
proftpd	88,876	155,908	43,670	8	228	4,633	4,869	10,559	4,667	5,633	4
lighttpd	15,044	27,788	9,466	7	217	598	814	8,215	3,303	2,817	5
nginx server	59,105	219,880	31,199	6	807	1,100	1,913	19,051	13,874	8,363	5
Other applications											
SoX 14.4.2	47,078	67,608	26,241	7	1,196	1,828	3,031	16,898	13,586	9,580	7
TinTin++ 2.01.6	34,367	34,431	17,725	7	465	2,023	2,495	8,908	1,130	4,593	4
dcraw 9.28	24,912	26,071	9,470	6	137	1,506	1,649	4,401	4,190	1,233	3
ngiflib 0.4	1,024	1,040	454	7	17	94	118	393	393	97	2
Gravity 0.3.5	36,439	53,220	22,185	7	1,415	537	1,959	10,806	6,277	4,645	4
MP3Gain 1.5.2	6,320	6,463	2,947	6	19	514	539	1,842	1,523	989	6
NASM 2.14.02	36,706	42,782	15,529	6	154	1,032	1,192	6,712	1,863	4,316	4
Jhead 3.00	4,813	4,927	2,127	6	18	495	519	1,605	827	679	5

Table 3: The internal statistics of SELECTIVETAINT for the tested benchmarks

1.10x-1.77x. SELECTIVETAINT performs even better with 1.12x-1.91x which outperforms `libdft` by 1.12x-2.08x.

6.3 Security Case Studies

Protecting `nginx` web server. To show that our tools could be used to detect real-world attacks, we first implemented a buffer overflow attack detector and used it to protect the `nginx` web server. To test its effectiveness, we generated an exploit based on the buffer overflow vulnerability CVE-2013-2028. By leveraging this vulnerability, an attacker could send a malformed request that triggers an integer signedness error which further causes a stack-based buffer overflow. This bug can be used in a denial-of-service attack or cause arbitrary code execution. Without any surprise, our SELECTIVETAINT detects the exploit at the `ret` instruction because the return value stored on the stack is tainted.

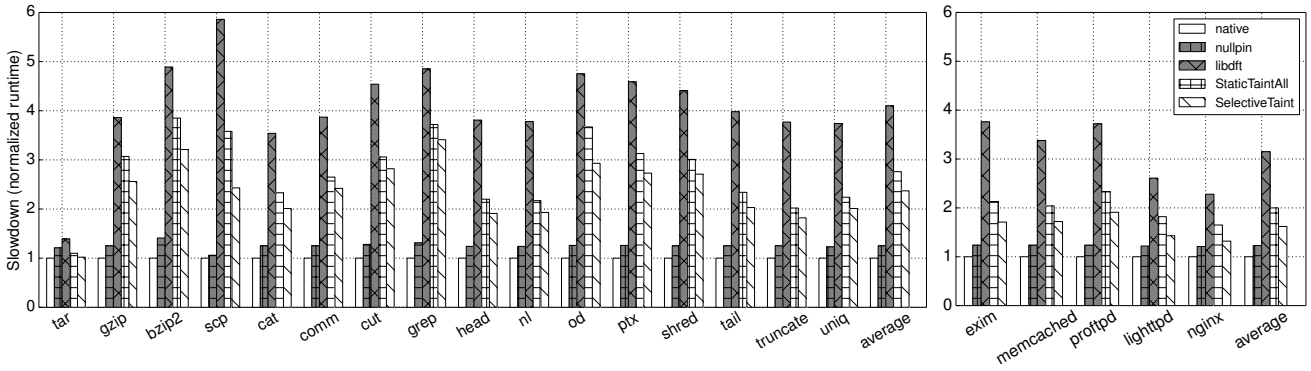
Protecting other binaries against recent memory exploits. We further tested eight recent real world software vulnerabilities from Common Vulnerabilities and Exposures (CVE)[1], which are listed in Table 4. The collected vulnerabilities covered a broad range of software vulnerabilities, including buffer overflow vulnerability, double free vulnerability, and integer underflow vulnerability, which manifested in varied programs such as sound processing utilities `SoX`, programming language interpreter `Gravity`, and audio normalization software `MP3Gain`.

We implemented the corresponding exploits to compromise these vulnerabilities and validate whether SELECTIVETAINT is able to detect the attacks. For instance, to exploit `CVE-2017-1000437` vulnerability in `Gravity 0.3.5`, we developed a malformed `gravity` programming language source code file, which overflowed the program stack to rewrite the return address with payloads in source code file. To exploit `CVE-2019-7629` vulnerability in `TinTin++ 2.10.6`, we set up a simple game server with exploits that keep sending crafted message which overflowed the multiplayer online game client `TinTin++`. Then, the tested binaries were instrumented with SELECTIVETAINT. In all cases, SELECTIVETAINT successfully detects the exploits which shows SELECTIVETAINT can facilitate real world vulnerability detection in various software.

7 Limitations and Future Work

Augmenting static analysis with dynamic information. As static analysis lacks dynamic information, SELECTIVETAINT has unknown values from multiple sources as shown in §4.2 and also VSA is an over-approximation of the possible values

[1] https://cve.mitre.org/



(a) UNIX utilities **(b)** Network daemons

Figure 7: The performance overhead of the tested benchmarks

Program	Category	Vulnerability	CVE ID	StaticTaintAll	SelectiveTaint
SoX 14.4.2	Sound Processing Utilities	Buffer Overflow	CVE-2019-8356	✓	✓
TinTin++ 2.01.6	Multiplayer Online Game Client	Buffer Overflow	CVE-2019-7629	✓	✓
dcraw 9.28	Raw Image Decoder	Buffer Overflow	CVE-2018-19655	✓	✓
ngiflib 0.4	GIF Format Decoding Library	Buffer Overflow	CVE-2018-11575	✓	✓
Gravity 0.3.5	Programming Language Interpreter	Buffer Overflow	CVE-2017-1000437	✓	✓
MP3Gain 1.5.2	Audio Normalization Software	Buffer Overflow	CVE-2017-14411	✓	✓
NASM 2.14.02	Assembler and Disassembler	Double Free	CVE-2019-8343	✓	✓
Jhead 3.00	Exif Jpeg Header Manipulation Tool	Integer Underflow	CVE-2018-6612	✓	✓
Nginx 1.4.0	Web Server	Buffer Overflow	CVE-2013-2028	✓	✓

Table 4: The tested vulnerable software and their vulnerabilities

each data object could hold. Dynamic information such as concrete values from run-time could further help SELECTIVETAINT. For instance, if we have some concrete inputs for a function, the binary rewriting could be tailored to those concrete inputs. In fact, recent taint improvement system Iodine [5] has used such a static and dynamic approach. Inspired by Iodine, we could similarly elide unnecessary taint propagation logic with the help from the dynamic information.

Context-aware instrumentation. Current binary instrumentation of SELECTIVETAINT is context-insensitive, i.e., SELECTIVETAINT instruments taint propagation logic disregarding the calling context of the binary function. However, we notice there could be cases that a function may need taint propagation in some contexts but not others (e.g., the function called from the beginning of program execution to the first taint introducing instruction). Therefore, it could be an improvement if we make the instrumentation context-aware or use multiple copies of the function (some contains taint, and some never). We plan to validate whether this could be a viable approach in one of our future works.

Improving static binary analysis. We have made several assumptions about the binary code to ease our binary analysis, e.g., we assume the code under analysis is not obfuscated and we do not consider dynamic generated code, since they are current obstacles and challenges for static binary analysis in general. That is, any improvement from static binary code analysis could benefit SELECTIVETAINT. In our implementation, we use `Dyninst` [7] and successfully rewrite all binaries in our evaluation without encountering corner cases described in `Dyninst`'s limitations. Future work may include implementations using other binary rewriters such as MULTIVERSE [6] and DDISASM [13].

Improving CFG reconstruction for a more precise alias analysis. As we have seen in the soundness analysis of SELECTIVETAINT (§4.3.2), improving the precision of alias analysis and CFG reconstruction can improve the soundness of SELECTIVETAINT. For instance, in instruction `call eax` where there could be aliasing between formal parameters and actual parameters, a precise alias analysis result would greatly affect the control flow targets and thus the CFG reconstruction. Though we use the approaches in §4.1 and §4.2 to improve the precision, a more precise alias analysis can largely benefit our analysis.

8 Related Work

Dynamic Data Flow Tracking (DDFT). Over the past decades, many DDFT systems were built. DYTAN [11] is one of the first such a tool that allows customized taint analysis, and it can also track implicit information flows due to control dependences. However, its performance overhead can be as high as 50x when performing dynamic taint analysis with both control- and data-flow based propagation and 30x for data-flow based propagation alone. Saxena et al. [32] propose

a static technique that recovers higher level structures from x86 binaries and apply it to the context of taint tracking. Unlike our approach, their *stack analysis* trades off analysis accuracy over performance, e.g., stack analysis ignores global and heap memory while VSA tracks global and heap memory. Also, Saxena et al.'s analysis mainly enables optimizations such as tag-sharing, but our analysis is the first static taint analysis to selectively instrument instructions. MINEMU [8] aims to design efficient emulator, using new memory layout and SSE registers to improve the taint analysis. Being a highly-optimized DDFT framework, `libdft` [17] shows a faster performance than previous efforts; for instance, `libdft` imposes about 4x slowdown for `gzip` when compressing or decompressing files.

Recent efforts [5, 15, 16, 24, 25] further improve the performance overhead of DDFT. For instance, Jee et al. propose *Taint Flow Algebra* (TFA) [16] that separates the program logic from data flow tracking, transforms the data flow tracking logic into an intermediate representation, and then performs classic compiler optimizations. SHADOWREPLICA [15] runs DFT in parallel in a shadow thread for each application thread and uses an off-line application analysis phase which leverages both static and dynamic analysis approaches to minimize the information needed to communicate between both threads. TAINTPIPE [25] uses *pipelined symbolic taint analysis* that both parallelizs and pipelines taint analysis. STRAIGHTTAINT [24] logs control flow profiling and execution state when taint seeds were first introduced. Most recently, IODINE [5] uses an optimistic hybrid analysis which restricts predicated static analysis to elide a runtime monitor only when it is proven to be a safe elision. Different from our approach, Iodine is built atop LLVM IR, which requires source code of the target application, whereas SELECTIVETAINT is a binary only approach.

Binary Rewriting. There is also a large body of work on static binary rewriting. Most recent efforts include UROBOROS [37], RAMBLR [36], MULTIVERSE [6], probabilistic disassembly [23], and DDISASM [13]. UROBOROS [37] is a tool which repeatedly dissembles the executable such that the generated code could be reassembled back to working binaries. RAMBLR [36] further analyzes the assumptions of UROBOROS and finds multiple complex corner cases that must be considered in symbolization. RAMBLR applies advanced static analyses, e.g., VSA, and achieves great performance for a static rewriter. MULTIVERSE [6] is the first static binary rewriting tool that systematically rewrites x86 COTS binaries without heuristics. Probabilistic disassembly [23] uses probabilities to model the uncertainty caused by the information loss during compilation and assembling. Features such as data flow and control flow features are leveraged to compute a probability for each address in the code space to indicate the likelihood to be an instruction. DDISASM [13] combines static analysis and heuristics in Datalog and shows that Datalog's inference process suits for the disassembling. In our implementation, while we could have used the most recent work such as MULTIVERSE, we use `Dyninst` [7] instead due to its rich APIs.

Alias Analysis on Binary. Prior efforts on binary alias analysis either introduce an IR and use Datalog to reason about points-to relations [9], or introduce sets for values held at each program point (e.g., abstract address sets [12], or symbolic value sets [1]). The alias relation of two variables is determined by whether the abstraction sets of these two variables intersect, e.g., intersection of abstract address sets [12], symbolic value sets [1], and points-to predicates results [9], respectively. Also, a number of earlier efforts (e.g., [1, 12]) do not further resolve indirect jumps in CFG and reconstruct more CFG edges and yet this limits the analysis precision. They also assume no system calls. However, system calls may introduce uninitialized value sets into the system, and the work by Debray et al. [12] uses less general sets of values, which is residue-based (module k), whereas we use all possible values.

9 Conclusion

We have presented an efficient static analysis based data flow tracking framework SELECTIVETAINT. Unlike previous taint analysis that uses dynamic binary instrumentation, SELECTIVETAINT is built atop static binary rewriting. The key insight is to use VSA to identify the instructions that never involve taint analysis, and then rewrite the rest to implement the taint analysis. We have tested SELECTIVETAINT with 29 binary programs including 16 Unix utilities, five network daemons, and eight vulnerable applications and observed a superior performance, which is 1.7x faster than that of the state of the art dynamic taint analysis tools.

Acknowledgment

We are grateful to our shepherd Vasileios P. Kemerlis as well as the anonymous reviewers including those from the artifact evaluation committee for their very constructive feedback. We also would like to thank Haohuang Wen for his assistance during the evaluation. This research was supported in part by DARPA award N6600120C4020, NSF awards 1750809 and 1834215, and ONR award N00014-17-1-2995.

Availability

The source code of SELECTIVETAINT and also the benchmark programs used during the evaluation have been made public available at https://github.com/OSUSecLab/SelectiveTaint.

References

[1] W. Amme, P. Braun, E. Zehendner, and F. Thomasset. Data dependence analysis of assembly code. In *Proceedings of the 1998 International Conference on Parallel Architectures and Compilation Techniques*, PACT '98, pages 340–347, Washington, DC, USA, 1998. IEEE Computer Society.

[2] Q. N. Anh. Capstone: Next generation disassembly framework. In *Proceedings of the 2014 Black Hat USA*, Black Hat USA '14, 2014.

[3] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 executables. In *Proceedings of the 2004 International Conference on Compiler Construction*, CC '04, pages 5–23, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[4] G. Balakrishnan and T. Reps. WYSINWYX: What You See is Not What You eXecute. *ACM Transactions on Programming Languages and Systems*, 32(6):23:1–23:84, Aug. 2010.

[5] S. Banerjee, D. Devecsery, P. M. Chen, and S. Narayanasamy. Iodine: Fast dynamic taint tracking using rollback-free optimistic hybrid analysis. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP '19, pages 712–726, 2019.

[6] E. Bauman, Z. Lin, and K. Hamlen. Superset disassembly: Statically rewriting x86 binaries without heuristics. In *Proceedings of the 25th Annual Network and Distributed System Security Symposium*, NDSS '18, San Diego, CA, Feb. 2018.

[7] A. R. Bernat and B. P. Miller. Anywhere, any-time binary instrumentation. In *Proceedings of the 10th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools*, PASTE '11, pages 9–16, New York, NY, USA, 2011. ACM.

[8] E. Bosman, A. Slowinska, and H. Bos. Minemu: The world's fastest taint tracker. In *Proceedings of the 14th International Symposium on Recent Advances in Intrusion Detection*, RAID '11, pages 1–20, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[9] D. Brumley and J. Newsome. Alias analysis for assembly. Technical report, Carnegie Mellon University, 2006.

[10] J. Caballero, H. Yin, Z. Liang, and D. Song. Polyglot: Automatic extraction of protocol message format using dynamic binary analysis. In *Proceedings of the 14th ACM conference on Computer and Communications Security*, CCS '07, pages 317–329. ACM, 2007.

[11] J. Clause, W. Li, and A. Orso. Dytan: A generic dynamic taint analysis framework. In *Proceedings of the 2007 International Symposium on Software Testing and Analysis*, ISSTA '07, pages 196–206, New York, NY, USA, 2007. ACM.

[12] S. Debray, R. Muth, and M. Weippert. Alias analysis of executable code. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 12–24, New York, NY, USA, 1998. ACM.

[13] A. Flores-Montoya and E. Schulte. Datalog disassembly. In *Proceedings of the 29th USENIX Security Symposium*, USENIX Security '20, pages 1075–1092. USENIX Association, Aug. 2020.

[14] W. G. J. Halfond, A. Orso, and P. Manolios. Using positive tainting and syntax-aware evaluation to counter sql injection attacks. In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, FSE '14, pages 175–185, New York, NY, USA, 2006. ACM.

[15] K. Jee, V. P. Kemerlis, A. D. Keromytis, and G. Portokalidis. ShadowReplica: Efficient parallelization of dynamic data flow tracking. In *Proceedings of the 20th ACM Conference on Computer and Communications Security*, CCS '13, pages 235–246, New York, NY, USA, 2013. ACM.

[16] K. Jee, G. Portokalidis, V. P. Kemerlis, S. Ghosh, D. I. August, and A. D. Keromytis. A general approach for efficiently accelerating software-based dynamic data flow tracking on commodity hardware. In *Proceedings of the 19th Annual Network and Distributed System Security Symposium*, NDSS '12, 2012.

[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 ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments*, VEE '12, pages 121–132, New York, NY, USA, 2012. ACM.

[18] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proceedings of the 20th Annual Network and Distributed System Security Symposium*, NDSS '13, 2013.

[19] Z. Lin, X. Jiang, D. Xu, and X. Zhang. Automatic protocol format reverse engineering through context-aware monitored execution. In *Proceedings of the 15th Annual Network and Distributed System Security Symposium*, NDSS '08, San Diego, CA, February 2008.

[20] Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Network and Distributed System Security Symposium*, NDSS'10, 2010.

[21] B. Livshits, M. Sridharan, Y. Smaragdakis, O. Lhoták, J. N. Amaral, B.-Y. E. Chang, S. Z. Guyer, U. P. Khedker, A. Møller, and D. Vardoulakis. In defense of soundiness: a manifesto. *Communications of the ACM*, 58(2):44–46, Jan. 2015.

[22] 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 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

[23] K. Miller, Y. Kwon, Y. Sun, Z. Zhang, X. Zhang, and Z. Lin. Probabilistic disassembly. In *Proceedings of the 41st International Conference on Software Engineering*, ICSE '19, page 1187–1198. IEEE Press, 2019.

[24] J. Ming, D. Wu, J. Wang, G. Xiao, and P. Liu. StraightTaint: Decoupled offline symbolic taint analysis. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering*, ASE '16, pages 308–319, New York, NY, USA, 2016. ACM.

[25] 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, pages 65–80, Washington, D.C., 2015. USENIX Association.

[26] P. Muntean, M. Fischer, G. Tan, Z. Lin, J. Grossklags, and C. Eckert. τCFI: Type-assisted control flow integrity for x86-64 binaries. In *Proceedings of the 21st International Symposium on Research in Attacks, Intrusions, and Defenses*, RAID '18, pages 423–444. Springer International Publishing, 2018.

[27] S. Nagy and M. Hicks. Full-speed fuzzing: Reducing fuzzing overhead through coverage-guided tracing. In *Proceedings of the 40th IEEE Symposium on Security and Privacy*, SP '19, pages 787–802, May 2019.

[28] J. Newsome and D. Song. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the 12th Annual Network and Distributed Systems Security Symposium*, NDSS '05, 2005.

[29] A. Nguyen-Tuong, S. Guarnieri, D. Greene, J. Shirley, and D. Evans. Automatically hardening web applications using precise tainting. In *Proceedings of the 2005 IFIP International Information Security Conference*, IFIP SEC '05, pages 295–307, Boston, MA, 2005. Springer US.

[30] T. Pietraszek and C. V. Berghe. Defending against injection attacks through context-sensitive string evaluation. In *Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection*, RAID '06, pages 124–145, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[31] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, Sept. 1994.

[32] P. Saxena, R. Sekar, and V. Puranik. Efficient fine-grained binary instrumentation with applications to taint-tracking. In *Proceedings of the 2008 International Symposium on Code Generation and Optimization*, CGO '08, page 74–83, New York, NY, USA, 2008. ACM.

[33] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, SP '16, 2016.

[34] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August. RIFLE: An architectural framework for user-centric information-flow security. In *Proceedings of the 37th Annual IEEE/ACM International Symposium on Microarchitecture*, MICRO 37, pages 243–254, Washington, DC, USA, 2004. IEEE Computer Society.

[35] V. van der Veen, E. Göktaş, M. Contag, A. Pawlowski, X. Chen, S. Rawat, H. Bos, T. Holz, E. Athanasopoulos, and C. Giuffrida. A tough call: Mitigating advanced code-reuse attacks at the binary level. In *Proceedings of the 37th IEEE Symposium on Security and Privacy*, SP '16, pages 934–953, 2016.

[36] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna. Ramblr: Making reassembly great again. In *Proceedings of the 24th Annual Network and Distributed System Security Symposium*, NDSS '17, 2017.

[37] S. Wang, P. Wang, and D. Wu. Reassembleable disassembling. In *Proceedings of the 24th USENIX Security Symposium*, USENIX Security '15, pages 627–642, Washington, D.C., 2015. USENIX Association.

[38] J. Whaley and M. S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[39] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda. Panorama: Capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM Conference on Computer and Communications Security*, CCS '07, pages 116–127, New York, NY, USA, 2007. ACM.

[40] J. Zeng, Y. Fu, K. 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, Berlin, Germany, Nov. 2013.

[41] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall. TaintEraser: Protecting sensitive data leaks using application-level taint tracking. *ACM SIGOPS Operating Systems Review*, 45(1):142–154, Feb. 2011.