One Engine To Serve 'em All: Inferring Taint Rules Without Architectural Semantics

Zheng Leong Chua[1†] Yanhao Wang[2,3†] Teodora Baluta[1] Prateek Saxena[1*] Zhenkai Liang[1*] Purui Su[2,3*]
[1]School of Computing, National University of Singapore
[2]TCA/SKLCS, Institute of Software, Chinese Academy of Sciences [3]University of Chinese Academy of Sciences
{chuazl, teobaluta, prateeks,liangzk}@comp.nus.edu.sg {wangyanhao, purui}@iscas.ac.cn

***Abstract*—Dynamic binary taint analysis has wide applications in the security analysis of commercial-off-the-shelf (COTS) binaries. One of the key challenges in dynamic binary analysis is to specify the taint rules that capture how taint information propagates for each instruction on an architecture. Most of the existing solutions specify taint rules using a *deductive* approach by summarizing the rules manually after analyzing the instruction semantics. Intuitively, taint propagation reflects on how an instruction input affects its output, and thus can be *observed* from instruction executions. In this work, we propose an *inductive* method for taint propagation and develop a universal taint tracking engine that is architecture-agnostic. Our taint engine, TAINTINDUCE, can learn taint rules with minimal architectural knowledge by observing the execution behavior of instructions. To measure its correctness and guide taint rule generation, we define the precise notion of soundness for bit-level taint tracking in this novel setup. In our evaluation, we show that TAINTINDUCE automatically learns rules for 4 widely used architectures: x86, x64, AArch64, and MIPS-I. It can detect vulnerabilities for 24 CVEs in 15 applications on both Linux and Windows over millions of instructions and is comparable with other mature existing tools (TEMU [51], libdft [32], Triton [42]). TAINTINDUCE can be used as a stand-alone taint engine or be used to complement existing taint engines for unhandled instructions. Further, it can be used as a cross-referencing tool to uncover bugs in taint engines, emulation implementations and ISA documentations.**

I. INTRODUCTION

Dynamic taint analysis is a form of information flow analysis, which tracks how certain initial "tainted" inputs exert influence on states during a program execution and detects if such tainted states are used in critical operations of a program. It has wide applications in security. For example, it is used for vulnerability detection or diagnosis [14], [23], [31], [38], [44], [48], [49], [53], [55], [56], [60], privacy analysis [13], [46], [61], and protocol recovery [16].

Dynamic tainting has been particularly useful for analyzing commercial-off-the-shelf (COTS) software in binary form. Since its introduction over a decade ago, numerous taint analysis engines have been developed. Most of these taint analysis engines have been based on a *deductive* approach [10], [20], [29], [32], [42], [51]. A taint engine has a set of static rules called taint propagation rules capturing how the inputs of a program statement influence (or taint) its outputs. These rules are typically specified once for a target processor architecture. When used for analyzing a program under concrete inputs, the taint engine evaluates these rules on the given inputs, instruction by instruction, until the taint information propagates to the point of interest. This approach, however, requires manual modeling of the target instruction set semantics. For an architecture like x86, instruction set descriptions span multiple volumes of text totaling thousands of pages [17], and each operation code has subtle variation in taint rules based on values of its operands. Developers typically choose to implement a subset of instructions commonly used and omit uncommon instructions, like floating-point or vector instructions. Recent articles show that optimized software detects CPU capabilities at runtime selecting uncommon instructions if possible [2]. Attackers can also take advantage of this weakness to evade detection using uncommon instructions [40].

Even for supported instructions, existing tools have human-engineered taint rules. Writing rules can take a massive effort in dealing with the intricate semantics and corner cases inherent in complex instruction sets. To illustrate the challenge, consider two instructions: `and eax,0x0` and `and eax,0xffffffff` on Intel x86. Both instructions have the same operation code; however, their taint propagation semantics differ substantially — the first one always sets taint for `eax` to zero whereas the latter preserves its output value. In the first instruction, the input has *no* influence on the output and the latter has complete influence. Due to this complexity, most taint engine implementations are far from being comprehensive and accurate. Several examples presented in Section II-B and our experiments confirm this. For instructions that are supported, in practice, a large number of declassification rules and exceptional (or idiomatic) rules have to be added incrementally. As software, compilers, instruction set specifications, and processor implementations evolve, creating and maintaining robust taint engines becomes burdensome.

For the above reasons, existing binary taint analysis tools are often developed individually for every target architecture. It is natural to ask: does there exist a "universal" algorithm that

[†] co-leading authors. [*] corresponding authors.

Network and Distributed Systems Security (NDSS) Symposium 2019
24-27 February 2019, San Diego, CA, USA
ISBN 1-891562-55-X
https://dx.doi.org/10.14722/ndss.2019.23339
www.ndss-symposium.org

can learn to perform taint analysis with little knowledge about the underlying architecture? Such a solution naturally enables taint analysis of an executable for any target architecture. As taint analysis is based on information flow, it suggests the possibility of a simpler, and hitherto unexplored, approach based on *inductive* inference. Instead of starting from a set of static rules for a target architecture, we *mutate* concrete values of inputs to an instruction and *observe* the changes to the output state: if changes in tainted inputs of an instruction change its outputs, then by definition this influence suggests that taint propagates. This approach has a significant advantage that it is mostly *agnostic* to the target architecture since the perturbation strategy does not need to understand the semantics of the operations being analyzed; it suffices to treat a program instruction as a black-box. Second, it only computes taint rules for instructions present in the program of interest, under the values provided. This eliminates pre-specification of rules for thousands of instructions under billions of program states possible, but not necessarily arising in the analyzed program.

Challenges. The inductive inference approach directly suggests a method: given an instruction, exhaustively enumerate values of tainted inputs and observe if output change. Consider the x86-64 instruction `and rax, rbx` wherein `rax` is the only tainted input and `rbx` is zero; the taint engine should not propagate taint for `rax` to its output. To discover this rule, a naive approach may try all 64-bit values possible for its tainted input `rax`. This method is *sound*, i.e., when the output is tainted, there is a pair of values that produce differing outputs, acting as witness to exhibiting the influence. The method is *complete*, i.e., it misses no such witnesses as it exhaustively searches all values for tainted inputs. However, it is intractably slow to enumerate a 64-bit space.

Approach. In this paper, we propose a novel approach to making such taint rule inference tractable and practical for binary code. Our approach makes almost no assumptions about the semantics of the underlying instruction set, beyond the ability to observe and mutate concrete inputs and outputs of instruction executions. The key idea is to sample and test an instruction behavior from a tractable number of input-output samples. Our approach infers a set of succinct rules for taint propagation on-the-fly, given a program and its inputs for analysis. Certain empirical characteristics of modern architectures make this approach feasible. First, most (but not all) instructions are either highly sensitive to changes in certain inputs or none at all. Both such instruction classes need a very small number of samples to determine if their tainted inputs impact output. Second, succinct taint rules learned from a small number of tested samples *generalize* well to capture the behavior of instructions on unseen samples. Finally, the number of unique instructions in a program is far smaller than the full set in an ISA or in its real execution traces, by orders of magnitude. Hence, learned rules can be *memoized* and applied without being re-calculated, providing efficiency.

To better understand this approach (and taint analysis techniques in general), we provide a formal definition of soundness and completeness that captures a precise notion of influence between program states during a concrete run of the program. This definition provides a novel perspective on the notion of under- or over-tainting observed in many prior works [47]; further, it allows one to empirically test the correctness of a taint engine implementation by concretely executing it. Second, we build a prototype tool called TAINTINDUCE that can operate in two modes: *exact* and *generalization*. In exact mode, TAINTINDUCE learns rules that never over-estimate the influence of program states at a point of the execution on another state. In generalization mode, TAINTINDUCE carefully trades off soundness for efficiency, by extending the applicability of its rules learned for an instruction on unseen program states optimistically.

Our TAINTINDUCE prototype is implemented on the `unicorn` [18] CPU emulator, which is built on `Qemu` and has support for 4 widely used architectures. TAINTINDUCE can serve as a stand-alone taint analysis engine, requiring minimal configuration per architecture or to complement existing taint engines for unhandled instructions. Further, it is useful as a *cross-referencing* tool to compare the correctness of other taint engines, emulator implementations, or even CPU implementations, evaluated from the abstraction of taint computation.

Evaluation Results. First, TAINTINDUCE is a feasible architecture-agnostic approach with considerable simplicity. It automatically learns taint rules for thousands of instructions correctly across 4 widely-used architectures: x86, x64, AArch64, and MIPS-I, with no specialized knowledge. We automatically check the rules for $1,527$ instructions, finding more than $1,062$ cases are sound even in TAINTINDUCE's generalization mode (in which soundness is not theoretically guaranteed). Second, TAINTINDUCE successfully detects vulnerabilities for 24 CVEs and tests 15 applications on both Windows and Linux, propagating taint over 8 million instructions in these benchmarks. Third, we find that TAINTINDUCE is comparable to the 3 popular dynamic taint analysis engines: `Triton` [42], `libdft` [32], and `TEMU` [51]. It propagates taint the same way as these tools in 93.27% of over millions of instructions in which its taint rules were applied.

In addition, TAINTINDUCE is useful as a cross-reference tool. TAINTINDUCE finds that existing tools propagate taint unsoundly (or over-taints) in roughly 6.64% of the millions of instructions handling taint data. It uncovers 17 missing instructions or wrongly emulated instructions in `unicorn`, one error in the Intel developer manual, one case of ambiguous documentation of the ISA specification, and one instruction which has differing CPU implementations. Despite rule generalization, TAINTINDUCE produces unsound rules for only 0.09% of the millions of instructions tested in concrete runs, an order of magnitude lower than other tested tools.

Contributions. We claim the following contributions:

- We propose a novel method, TAINTINDUCE, to perform dynamic taint analysis. To the best of our knowledge, TAINTINDUCE is the first to perform dynamic taint analysis for binary code requiring minimal architectural semantics, which is through on-the-fly taint rule inference.
- We precisely define influence, soundness, and completeness for dynamic taint analysis of program executions. We provide a sound algorithm to infer taint rules in a novel setup and show how to generalize systematically.
- We implemented TAINTINDUCE and evaluated with millions of instructions. We empirically show that our approach is useful as a stand-alone taint engine and a cross-referencing debugging aid. It is applicable to 4 processor

architectures with minimal specialized knowledge, finding a multitude of errors in existing state-of-the-art tools, emulators and ISA software developer manuals.

II. Problem

In dynamic taint analysis of binary code, taint information is tracked for each bit (or some specified granularity) of program state (e.g. register or program memory) during a concrete execution of the program. The taint information can be used in different applications. For instance, to reason about confidentiality properties, a taint-analysis application may mark confidential inputs as tainted and enforce a security check not permitting tainted values to be used in public outputs. Alternatively, to reason about program integrity, a taint-analysis application may configure the taint analysis to treat inputs controlled by remote adversaries tainted and not permit tainted values to be used in critical control-flow transfers. We work in this standard setup of taint analysis, wherein the *source* and *sink* operations — locations where taint bits originate and are checked respectively — are an external specification [58].

A. The Taint Inference Problem

We wish to design a universal taint tracking mechanism for an architecture with minimal knowledge of the architecture's instruction semantics. The new taint tracking mechanism takes as input an executable program binary and a concrete set of inputs under which the binary is analyzed. For each bit in the program state, the tainting mechanism maintains a corresponding taint metadata bit in a data structure that we call *taint map*. It computes taint bits by single-stepping each instruction in the concrete execution of the program under the provided inputs. We assume that the program (or the execution platform) has been instrumented in advance to initialize taint bits at the analyst-specified source locations (e.g. at points where network input is read), and to check them at sinks (e.g., at indirect control flow transfers). This taint tracking mechanism differs from existing taint analysis mechanisms [38] in that it automatically *infers taint rules* instead of requiring taint rules to be specified based on architectural knowledge.

Taint rules decide how input states of an instruction affect its output states. Intuitively, we wish to interpret taint tracking as capturing the notion of *influence*: if an independent change to the input value of a bit x causes a change in the output bit y, as a result of executing an instruction, we say that x influences y via that instruction. This interpretation enables a new way to perform taint tracking, based on observations of system states rather than pre-specification of instruction behavior.

Execution Model & Assumptions. We assume that our taint engine can *mutate* program state and *observe* the effect of this mutation for a given instruction, at a point in the program execution. Based on such input-output observations, it infers which program state bits should have their corresponding taint bits set or cleared. We learn one taint rule per instruction, making only the following set of assumptions:

- We assume a standard von Neumann architecture for which the taint engine can recognize instructions, and distinguish its register and memory accesses.
- We assume that the number of registers and memory slots are known.

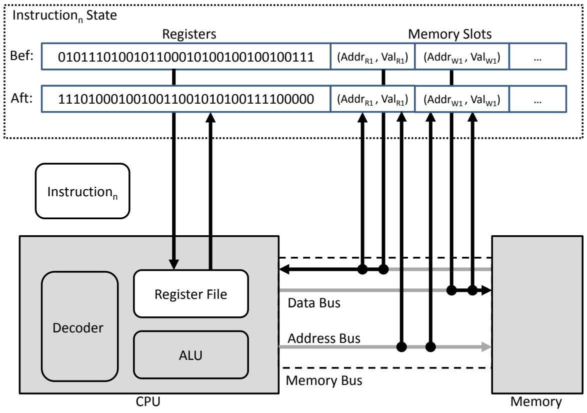


Fig. 1. Overview of program state observation structure and architecture assumptions. Shaded portions are assumed to be a black-box. Each memory observation is represented as a memory slot which is a tuple made up of address and value. The subscript $\mathbf{R}i$ represents the i^{th} read memory slot and $\mathbf{W}i$ represents the i^{th} write memory slot.

- We assume that the taint engine has a concrete evaluator that reads / writes registers and memory, as is possible with standard debuggers. Specifically, during a concrete program execution, the evaluator can intercept the read and written values of the next instruction. The evaluator runs its mutation or probing as a shadow computation, without affecting the original execution of the program under provided inputs. Our strategy to implement such in-vitro shadow computation is presented in Section IV.

Figure 1 captures our assumptions about the class of architectures it is applicable to, and the level of access our approach needs precisely. In particular, our problem definition does *not* require the following assumptions:

- We do not need to know specialized semantics of registers and in fact, we treat the entire register set as a vector of bits. For generality, we assume that each value of interest is of a known bit-width k, say $k = 32$ (for x86) or $k = 64$ (for x64). If analysts do not wish to specify it, a large enough bit-width can be assumed, and the inferred taint rules will implicitly operate on the right bit-width.
- We do not need to be able to disassemble the instruction, two instructions with the same operation code but different operands (say `mov eax, 0x0` and `mov eax, 0x1`) are treated as two different instructions.
- We make no assumptions about the program compilation, optimization, OS-specific ABIs and software interfaces.

Soundness & Completeness. Having discussed the semantics of taint tracking syntactically, we aim to show a new approach to binary taint analysis and compare to existing works; thus, it is important to define the soundness or correctness criterion precisely. We would like to define soundness as a property of a taint tracking system, which ensures that it never over-approximates to include influences which do not exist, i.e., it does not over-taint. A taint rule system that never sets taint is trivially sound. Therefore, defining *completeness* is important in measuring the correctness of a solution.

In this work, we define soundness (and completeness) under a general instruction definition: an instruction is an unknown function operating on the entire program state at any point in the program execution. The program state is a bit vector of finite size. A program execution is simply a sequence of instructions evaluated, changing the program state from one value to another; our definitions of influence and soundness extend naturally to a sequence of instructions executed.

Let $I : \{0,1\}^n \rightarrow \{0,1\}^n$ be the instruction, modeled as an unknown function mapping one program state value to another. Let $S : \{0,1\}^n$ be a specific value of the program state before instruction I; $S[a]$ be the value of the bit location a in S; and let $\texttt{flip}(S,a)$ return the value S with the value of bit a flipped (or inverted). We say that $\langle I,S,x,y\rangle$ are in the *influence relation* Inf, if and only if $I(\texttt{flip}(S,x))[y] \neq I(S)[y]$. Let $T : \{0,1\}^n$ be a taint map. The taint rule of I is a function mapping the taint map before I's execution to the taint map after I's execution, based on I's semantics and values in S, $R_I : \{0,1\}^n \times \{0,1\}^n \rightarrow \{0,1\}^n$. We say that a taint engine is *sound* if:

$$R_I(S,T)[j] \implies \exists i, S|T[i] \wedge (\langle I,S,i,j\rangle \in Inf)$$

We say the engine is *complete* if:

$$\exists i, S|T[i] \wedge (\langle I,S,i,j\rangle \in Inf) \implies R_I(S,T)[j]$$

Note that our soundness/completeness definitions are new and non-classical; they are defined only with respect to a set of input states. They do *not* assume access to the ground truth about the behavior of an instruction on all possible inputs. We present a system that learns sound and complete taint rules with respect to a set of *observed* states (in its exact mode). Our notions further allow us to precisely compare soundness of any two taint systems on a given program execution. Testing of any taint system for soundness, as we define, can be automated by checking if bit-flips in an input changes instruction outputs.

Our eventual goal is not to have a perfectly sound and complete taint rule system, but rather a practical system that works well empirically. One could define the ground truth semantics of an instruction set for all possible states, and prove the above properties. In fact, recent work on the DECAF engine has manually specified instruction semantics for the integer arithmetic subset of instructions in SMT theories and has proved comparable properties of manually engineered taint rules [29]. While it is a laudable goal, scaling this approach to entire complex instruction sets (e.g. SIMD and FPU instructions) is infeasible as it requires human expertise and it relies on SMT solvers which have limited reasoning power for complex theories.

B. Challenges in Taint Inference

The central challenge is striking the right balance between soundness and completeness. To illustrate, let us revisit the example of a commonly used operation in x86, namely `and op1, op2` where the operands `op1` and `op2` can be 32-bit, 16-bit or 8-bit. The sound and complete rule for byte-level tracking is shown in Code 6[1].

Input context matters. Code 6 shows the complexity of handling the `and` instruction correctly on x86. As observed in Section I, a sound taint engine should examine the rule to apply in the context of a specific input value. The rules for this instruction are unsound (or incorrect) in 2 state-of-the-art taint analysis engines written by human developers. For example, `libdft` [32] implements byte-level tainting with the rule shown in Code 1. This states that if register `eax` is tainted and register `ebx` is not tainted then the taint of register `eax` is left unchanged. However, if the value of register `ebx` is `0` and `ebx` is untainted, then the taint of register `eax` should be cleared and it is not. This rule is unsound as it leads to over-tainting. `Triton` [42] implements a similar rule (Code 2) for register-level tracking which over-taints because it does not clear the taint when `ebx` is 0. Often, hand-engineered rules do *not* specialize the propagation rule based on input value contexts. However, our tool can automatically learn a rule that does not over-taint (Code 3).

```
if (True) {
  T[eax][0:8]   = T[eax][0:8]   | T[ebx][0:8];
  T[eax][8:16]  = T[eax][8:16]  | T[ebx][8:16];
  T[eax][16:24] = T[eax][16:24] | T[ebx][16:24];
  T[eax][24:32] = T[eax][24:32] | T[ebx][24:32];}
```

Code 1. `libdft : and eax, ebx`

```
if (True) { T[eax] = T[eax] | T[ebx];
  T[eflags][pf] = T[eax] | T[ebx];
  T[eflags][zf] = T[eax] | T[ebx];
  T[eflags][sf] = T[eax] | T[ebx];
  T[eflags][cf] = 0; T[eflags][of] = 0;}
```

Code 2. `Triton : and eax, ebx`

```
for (int x = 0; x < 32; x++)
  if (!ebx[x])  T[eax][x] = 0;
...
T[eflags][cf] = 0;    T[eflags][pf] = 0;
T[eflags][zf] = 0;    T[eflags][sf] = 0;
T[eflags][tf] = 0;    T[eflags][df] = 0;
```

Code 3. TAINTINDUCE (x86) : `and eax, ebx, T[ebx]=0` (Sound)

Incompleteness. In addition, existing tools are not only unsound but they are also incomplete. Flags are entirely missing in `libdft` which means the tool under-taints. If the second operand is an immediate value, `libdft` implements the rule in Code 4. When the immediate value is `0xff` and the first byte of `eax` is tainted, `PF`'s taint value should be set but libdft will never report it as tainted. Notice that the same rule is also over-tainting because it is tainting the upper 3 bytes in `eax`. We automatically learn a rule capable of handling taint propagation for flags and immediate values that is sound and complete (Code 5).

```
if (True) { T[eax][0:8]   = T[eax][0:8];
            T[eax][8:16]  = T[eax][8:16];
            T[eax][16:24] = T[eax][16:24];
            T[eax][24:32] = T[eax][24:32];}
```

Code 4. `libdft : and eax, 0xff`

```
if (True) { T[eax][0:8]  = T[eax][0:8];
  T[eax][8:32] = 0 ;  T[eflags][pf] = T[eax][0-8];
  T[eflags][cf] = 0;  T[eflags][zf] = T[eax][0-8];
  T[eflags][af] = 0;  T[eflags][tf] = 0;
  T[eflags][sf] = 0;  T[eflags][df] = 0;}
```

Code 5. TAINTINDUCE (x86) : `and eax, 0xff` (Sound)

[1] All examples throughout the paper are shown with register and flag names with suitable C-style operators for readability. Our approach does not know individual register names or widths, but rather learns over the range of bits in the register program state; further our analysis learns an internal, but equivalent representation, not in a C-style syntax. We use the $[i : j]$ notation to denote bit ranges corresponding to the interval $[i, j)$, e.g. `[0:8]` are bits 0 to 7. The $[i - j]$ notation stands for OR-ing the bits in $[i, j)$.

These examples show that practical errors in hand-engineered taint rules are commonplace. Our experiments quantify the rate of these errors in Section V-C.

Architectural quirks. Further, architectural quirks make hand-engineering rules extremely challenging. Code 6 shows that the semantics for the `and` instruction differs substantially, and perhaps non-intuitively on x64. Specifically, the 32-bit operand version (`and eax, ebx`) on x64 architecture will zero-extend the destination register. However, for the 16-bit and 8-bit operand version (`and ax, bx`), it will leave the 48 or 56 most significant bits untouched in the destination register [2]. None of the 3 taint analysis tools considered in our evaluation (`TEMU` [51], `libdft`, `Triton`) supports x64.

```
// t1 is the taint of op1; t2 is the taint of op2
// size is the size of the operands
// mode64bit is true if it operates in 64-bit mode
if (size == 64 || size == 32 || size == 16) {
  for (x = 0; x < size / 8; x++) {
    if (t1[x] & t2[x]) t1[x] = 1;
    else if (t1[x] and !t2[x])
      t1[x] = t1[x] & op2[x];
    else if (!t1[x] & t2[x])
      t1[x] = t2[x] & op1[x];
    else t1[x] = 0;
  } else if (size == 8) {
// 0 if it's lower 8 bits, 1 if it's upper 8 bits
    pos1 = isUpper(op1); pos2 = isUpper(op2);
    if (t1[pos1] & t2[pos2]) t1[pos1] = 1;
    else if (t1[pos1] & !t2[pos2])
      t1[pos1] = t1[pos1] & op2[pos2];
    else if (!t1[pos1] & t2[pos2])
      t1[pos1] = t2[pos2] & op1[pos1];
    else t1[pos1] = 0;}}
if (mode64bit == 1)
  for (x = 32; x < size; x++)  t1[x] = 0;
```

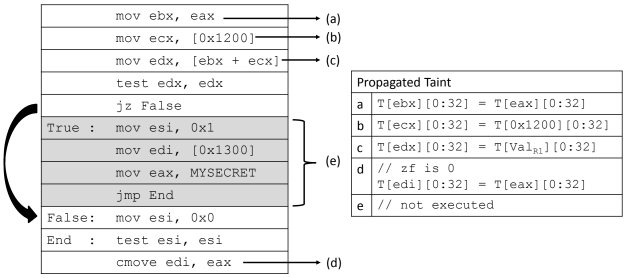
Code 6. `and op1, op2` for all bit widths (Complete and Sound)

III. Design

Our soundness / completeness definitions are agnostic to the taint propagation policy adopted by the taint engine. We first clarify the taint policy considered in this work. It is a standard policy used by most taint tracking systems, and our techniques can be tailored to suit other policies. We then present our design, which is simple and universal across all instructions and architectures reported in this work.

A. Taint Propagation Policy

A taint analysis client may decide different *propagation policies* based on its goals and demands on precision [35], [47]. We adopt a standard taint propagation policy used in most prior works. A common dynamic taint propagation policy is to track direct dependencies. Our policy further includes conditional dependencies between inputs and outputs of an instruction and a standard form of memory indirect dependencies. Specifically, our policy propagates taint from tainted memory values to assigned registers; however, a tainted memory *address* does *not* taint its values/content by default. That is, a read from a tainted address returns a tainted value only if the memory content is tainted, irrespective of the taint status of the pointer.



```
        mov ebx, eax
        mov ecx, [0x1200]
        mov edx, [ebx + ecx]
        test edx, edx
        jz False
True :  mov esi, 0x1
        mov edi, [0x1300]
        mov eax, MYSECRET
        jmp End
False:  mov esi, 0x0
End  :  test esi, esi
        cmove edi, eax
```

Propagated Taint	
a	T[ebx][0:32] = T[eax][0:32]
b	T[ecx][0:32] = T[0x1200][0:32]
c	T[edx][0:32] = T[Val$_{R1}$][0:32]
d	// zf is 0 T[edi][0:32] = T[eax][0:32]
e	// not executed

Fig. 2. Assembly snippet showing instructions of different influence types. Grey'ed instructions are not executed. (a)-(d) are explicit dependencies while (e) are implicit dependencies. (a) is a direct dependency, (b) is a direct dependency on the memory value. (c) is an indirect memory dependency on the address(ebx, ecx). (d) is a conditional indirect dependency on `eflags`. Our taint policy implies the shown rules.

For conditional dependencies, our policy states that taint is not propagated from the values that are conditioned on, but only from other inputs. This is a standard policy adopted in prior works to avoid over-tainting due to known challenges [12], [47], [50]. Like most taint systems, implicit flows are left untracked, as they reason about program logic on unexecuted control paths.We explain these notions with examples.

Following pioneering work by Denning [19], prior works define dependencies often in terms of program representation (assignments, if-else, loops). Since we do not have access to the program and consider each instruction as a black-box, we explain notions of direct and indirect dependencies based on influence observations. Direct dependencies are those influence relations that are same across all input-output observed states; see (a)-(b) in Figure 2. Simple register arithmetic and assignment instructions, for instance, create direct dependencies. Indirect dependencies are influence relations that change across different observations depending on some value of the input state; see (c)-(d) in Figure 2. All memory de-references and conditional statements are examples of indirect dependencies. In example (c), the taint status of register pointer does not affect the output taint. In example (d), `eflags` is conditioned on, and as per our standard policy, its taint status does not affect the outputs. All direct and indirect dependencies are *explicit*, i.e. they are observed along the analyzed path. Any dependencies that are not observed along the execution path are considered implicit; see (e) in Figure 2.

B. Overview

We build a tool called TaintInduce which is a new approach to bit-level tainting. The design of TaintInduce is outlined in Algorithm 1. Shown in the function `TaintProp`, TaintInduce takes as input the execution trace of a program run with some concrete input values and a taint map, initialized with taint sources. It iterates through the instructions in the program's execution. At any given point in the concrete execution of a program, our goal is to learn a rule (function `ruleInfer`) that propagates taint from the inputs to the outputs of the next instruction, under the specific program state at that execution point (Lines 3-16, Alg. 1). The taint map stores taint values for each register and memory accesses throughout the execution of the trace. It is updated after the execution of each instruction according to the inferred

[2] These semantics are due to the backward compatibility differences between 32-bit ISA versions and its predecessors, which differs from the 64-bit version of the ISA.

taint rule for the concrete input value (Line 14, Alg. 1). The applyRule function can be modified to propagate taint according to a different taint policy.

Program State. TAINTINDUCE learns taint propagation rules from observations on program states (Line 7 in TaintProp). The concrete program state consists of the processor register set and simulated memory accesses (Figure 1). A simulated memory access contains the memory address and its corresponding memory value. In TAINTINDUCE, we abstract the program state as a fixed-size bit-vector. The size is determined by the number of registers and a preset number of simulated memory access slots. Note that the number of simulated memory slots corresponds to the maximum possible number of memory operands that can be accessed in one instruction. When memory is read/written, the address and value of the simulated memory access are updated (see Section IV).

To analyze an instruction, TAINTINDUCE observes its behavior on a set of seed states. Generation of these seed states is described in Section IV. TAINTINDUCE systematically mutates the seed state via single bit flips, generating different input values to observe the behavior of the instruction (Lines 6-8, Alg. 1). From these observations, it learns a set of taint rules for computing $R_I(S,T)[j]$. The taint rules have the template:

$$\textbf{if } (\phi_j(S)) \textbf{ then } R_I(S,T)[j] := \vee_{i \in M_{I,S,j}} T[i]$$

$$\textbf{else } R_I(S,T)[j] := 0$$

where S is the program state bitvector and $M_{I,S,j}$ is a subset of the input bits that influence the j-th bit in the output state, $M_{I,S,j} = \{x | \langle I, S, x, j \rangle \in Inf\}$, learned by the engine and compactly represented as the rule set in the ruleInfer function. Effectively, the taint status of bit j is either the bitwise OR of taint status of a subset of input bits, or it is cleared. The task of our algorithm reduces to learning $M_{I,S,j}$ and the pre-condition ϕ_j to propagate the taint.

Notice that this approach learns rules only for those instructions that are concretely executed for the given program and its inputs. The learned rules are *memoized* in the ruleDB table and are directly applied (Lines 13-14, Alg. 1) when an instruction occurs again, provided a learned pre-condition ϕ_j is satisfied. TAINTINDUCE can export these rules for an architecture, and use them to analyze other programs on that architecture. Therefore, it is feasible to learn a working set of rules over time, sufficient for many practical applications.

Key Ideas. Our approach is novel in that it is designed to adhere to soundness as a yardstick and only deviates from it in a controlled way. One key idea in our approach is learning a rule that is *specialized* to the input context (or state value) observed. In its simplest form, the condition ϕ_j can capture a rule that is only valid to use in the specific program state that the execution is in. As TAINTINDUCE observes more program states on which an instruction is evaluated, it can expand the condition ϕ_j. We outline a way to do this *without* losing soundness — the applied rule would capture the notion of influence defined without over-approximating the influence relation. When operating in this way, we say that TAINTINDUCE is operating in exact mode. Learning the condition ϕ_j correctly is critical since it dictates when to clear the taint for an output bit. Setting taint to 0 is always sound since it can only lead to under-tainting (under-approximating the actual influence relation); however, ϕ_j must not miss cases when taint should be cleared, otherwise we risk over-tainting. We provide a procedure that achieves this goal in Section III-D.

Algorithm 1 TAINTINDUCE taint propagation.

```
 1: function TAINTPROP(trace, T, gen)
        // $trace$ - Execution trace of the program
        // $T$ - Taint map $T$ (with possible taint)
        // $gen$ - Generalization mode
 2:     ruleDB ← []
 3:     foreach instr∈trace do
 4:         seeds ← concInput ∪ GENRANDINPUTS()
 5:         obs ← []; rule ← []; $\phi$ ← []
 6:         foreach S∈seeds do
 7:             obs ← obs ∪ {o | o = ($S$,$i$,$E$,$E_{flip}$), $S_{flip}$ ← FLIP(S,$i$), $E$← EXECUTE(instr,$S$), $E_{flip}$ ← EXECUTE(instr,$S_{flip}$) for 0≤$i$<$n$}
 8:         end for
 9:         change, noChange ← gatherObs(obs)
10:         for $j \in 0$ to $n$ do
11:             rule[$j$],$\phi$[$j$]←RULEINFER($j$,change,noChange,gen)
12:         end for
13:         ruleDB[instr]←($\phi$, rule)
14:         APPLYRULE($\phi$,rule,concInput,T)
15:     end for
16: end function

17: function GATHEROBS(obs)
18:     change,noChange←[]
19:     foreach o ← (seed,$i$,seedOut,mutOut) ∈ obs do
20:         for $j \leftarrow 0$ to $n$ do
21:             if seedOut[$j$] ≠ mutOut[$j$] then change[$i$][$j$] ← o
22:             else noChange[$j$][$j$] ← o
23:             end if
24:         end for
25:     end for
26:     return change, noChange
27: end function

28: function RULEINFER($j$, change, noChange, gen)
29:     rule ← []; $\phi$ ← []
30:     for $i \leftarrow 0$ to $n$ do
31:         rule ← rule ∪ $i$
32:         truthTable[$True$] ← change[$i$][$j$]
33:         truthTable[$False$] ← noChange[$i$][$j$]
34:         C←COMPLEMENT(change[$i$][$j$]∪noChange[$i$][$j$])
35:         if gen then truthTable[$DontCare$] ← C
36:         else truthTable[$False$] ← truthTable[$False$] ∪ C
37:         end if
38:     end for
39:     $\phi$ ← BOOLMIN(truthTable)
40:     return rule, $\phi$
41: end function
```

Our second key insight is that TAINTINDUCE can generalize beyond the behaviors observed. In this *generalization* mode, TAINTINDUCE does not guarantee soundness; however, it learns rules that are more complete, which can be memoized and applied in larger program states. This yields better performance since memoized rules are applied more often. A key empirical discovery is that even when TAINTINDUCE operates in generalization mode (which is the default), it does not lead to excessive unsoundness and the rules learned work soundly in our experiments. Also, the learned rules do not under-taint excessively, and successfully work in detecting taint-style vulnerabilities in a number of real-world experiments.

Our approach is able to recover precise conditions ϕ_j which capture both direct dependencies as well as indirect dependencies, such as memory indirect and control dependencies. The specific propagation rules learned are for a policy we fix, as outlined in Section II. The key idea to recover such dependencies is to determine whether a bit i propagates taint to bit j unconditionally, i.e. independent of the values of other bits, as is the case with direct dependencies. When an instruction exhibits one kind of influence from bit i to bit j under certain conditions, and another kind of influence otherwise, this is a form of conditional dependence. We automatically learn these dependencies using an approach that works well in practice.

C. Modeling Direct Dependencies

Taint rule inference is achieved by observing the influence of the input bits on an output state. To do that, we generate a set of seed states as described in Section IV. For each seed state S of n bits, TAINTINDUCE generates n new mutated states by flipping each bit sequentially in S. Then, it concretely executes the instruction under these n mutated states and records the input-output states in an observations table, `obs`.

For a pair of bits (i, j), if a flip in the input i causes a change in output j (recorded in `change`), we propagate taint of input bit i to output bit j. Therefore, the output $R_I(S, T)[j]$ is the bitwise-OR of the taint of all bits which unconditionally influence j. Conversely, if changes in all input bits exhibit no change in the output bit j (recorded in `noChange`), we clear the taint for output j. For direct dependencies, when `change` contains observations, then `noChange` will be empty, and vice versa. Notice that we examine the change of values of each input bit i on itself during this process. Specifically, if bit value j is only influenced by itself, and no other bits, we would update $R_I(S, T)[j] := T[j]$. Since such a taint update is redundant, we eliminate it as an optimization. This case happens very often since for values that the instruction does not read or write, a change in its input value will reflect to its output value after the instruction is executed.

We point out that this taint rule inference is extremely simple, but powerful, since a large number of instructions exhibit their influence characteristics in single-bit-flip mutations. Further, the rule described above preserves soundness. When it sets the taint status of a bit to 1, we have a clear witness that a particular input bit has influenced it (as defined in Section II). When there is an invariance in the value of a bit with respect to changes in all input bits, we conservatively set it to zero — this is sound since it conservatively eliminates possible over-tainting in an output bit. Lastly, observe that the inferred taint rule to propagate taint from bit i to j is only valid under the concrete input state values for those tested for; nothing can be deduced about the instruction behavior on unobserved states.

TAINTINDUCE learns a succinct pre-condition (ϕ_j) for applying the inferred taint propagation rules. In our work, ϕ_j is a boolean formula in disjunctive normal form (DNF) over n variables denoting the bits of the input program state. For soundness, it suffices that ϕ_j be satisfied only by concretely observed states. To synthesize ϕ_j, TAINTINDUCE employs a procedure (outlined in Section III-D) that takes the observation set, the index of the output bit j, and returns ϕ_j which is satisfied by elements of the observed set.

Example: Direct Dependency (Sound). Consider the x86 instruction `and eax, 0xff` that we discussed in Section II-B. The `ruleInfer` algorithm collects observations where flipping a bit in register `eax` results in a change in the output register `eax`. In this example, TAINTINDUCE used 100 random seeds and observed 251 distinct input values out of the possible 256. `ruleInfer` produces the result that there is an influence from bit i in the input to bit i in the output across all the observed samples. Instead of the disjunction of all observed inputs, ϕ is of the more concise form seen in Code 7.

```
if ((!eax[0]&!eax[6]) || (eax[1]&eax[5])  ||
    (!eax[0]&!eax[3]) || (eax[3]&eax[4])  ||
    (!eax[3]&!eax[6]) || (eax[2]&!eax[5]) ||
    (!eax[4]&!eax[5]&eax[7]) || (eax[2]&eax[3]) ||
    (eax[0]&!eax[2]&eax[5])  || (eax[6]&!eax[7])||
    (eax[0]&!eax[1]&!eax[4]))
  T[eax][0:8] = T[eax][0:8];
```

Code 7. Exact mode - TAINTINDUCE (x86) : `and eax, 0xff` (Sound)

D. Learning Succinct Conditions

We now explain how TAINTINDUCE learns ϕ_j given an output bit j, and a set of program states observed (say Σ). The goal is to learn a succinct DNF-formula over n boolean variables signifying the program state bits, which is satisfied by values in Σ. TAINTINDUCE takes a function minimization approach to learning such a DNF formula. Specifically, we construct a function over the n bits of the program state, that returns `True` for all state values in Σ, and returns `False` otherwise. Conceptually, we can construct the truth table for such a function by setting the rows corresponding to values in Σ as `True` and remaining all rows to `False`. Then, we can invoke a boolean function minimization procedure over this truth table to obtain the equivalent DNF formula.

Boolean function minimization is a well-studied problem of finding the smallest boolean formula that is equivalent to a given function. That truth table does not need to be specified in enumerative form; it suffices to provide the entries evaluating to true and stating that all other entries should be treated as `False`. The problem is known to be NP-complete [54] for two-level boolean circuits. A classical procedure known as the Quine-McCluskey (QM) algorithm [36] produces the minimal possible representation. However, it has running time exponential in the number of input bits and as such, does not scale to hundreds of bits as in register state of modern architectures. Instead, we use the ESPRESSO [8] algorithm which is a greedy, heuristic-based algorithm that runs fast and produces solutions that are equivalent to the given input function. ESPRESSO does not guarantee a minimal form; however, it eliminates redundant clauses in the DNF form and in practice, the formulae it produces are fairly concise.

Note that ESPRESSO introduces no unsoundness. It returns a smaller representation of the function we construct, which exactly captures (by returning `True`) only for the elements of Σ — it does *not* learn any approximation or non-equivalent DNF form of the requested function. It only trades off succinctness for better efficiency compared to the QM algorithm.

E. Modeling Indirect Dependencies

The most common case of indirect dependencies is conditional dependencies — where the instruction exhibits multi-modal behavior conditioned on the values of some inputs.

Conditional dependencies. An *ambiguity*, or multi-modal behavior, happens if flipping bit i cause a change in bit j only for a subset of program state values. To handle conditional dependencies, TaintInduce has to identify what are the conditions which resolve the ambiguity. An example of this is the x64 conditional assignment instruction `cmovg rax, rbx`. It only assigns the value of `rbx` to `rax` if the value `rbx` is greater than zero, i.e. when zero, sign and overflow flag registers have specific values satisfied by the condition ($\texttt{ZF=0} \wedge \texttt{SF=OF}$). When the condition is not satisfied, the instruction does not perform the assignment.

When the taint rule to compute $R_I(S,T)[j]$ from $T[i]$ is ambiguous[3], TaintInduce groups observations based on if a change is observed or not (function `gatherObs`). For pairs of bits (i, j), TaintInduce learns a succinct condition ϕ_j for which $R_I(S,T)[j] := T[i]$ and $R_I(S,T)[j] := 0$ otherwise. The approach to learn ϕ_j relies on function minimization and is similar to the case for direct dependencies. The difference is that we learn ϕ_j for the subset of observed states where we observe an influence from i to j. Specifically, we construct a function over all n bits of the program state that returns `True` for all state values which we observed a change in j due to a change in i and `False` otherwise. A minimized boolean DNF formula can be obtained by invoking the procedure defined in Section III-D. Note that this procedure outlined is sound. When a taint rule that could propagate taint bits is applied, the learned pre-conditions ϕ_j capture exactly the set of state mutations which are observed in our test. Since boolean minimization ensures equivalence with the original function, $\overline{\phi_j}$ covers all unobserved inputs and clears $T[j]$, avoiding over-tainting.

Example: Conditional Dependence. Consider the x86 bitwise shift instruction `shl eax, cl` which shifts `eax` with a number of bits specified by the value of `cl` masked to 5 bits. As such, `shl` exhibits multiple behaviors depending on the value of `cl`. For example, if the masked value of `cl` is 0 then the taint status of `eax` remains unchanged. The taint rule for this behavior corresponds to the first branch of the `if` statement in Code 8. TaintInduce soundly infers the conditions under which this behavior applies as the subset of the observed samples where the lower 5 bits of `ecx` are set to 0. If the masked value of `cl` is 1, the taint value of each bit at index i of `eax` depends on the input taint value of the bit at position $i+1$ of `eax`. The conditions represent the subset of observed samples where the masked value is 1. This preserves soundness as taint propagates only on observed behaviors.

```
if (ecx == 0b01110101110011100010001111100000
    ...
    || ecx == 0b11101100110001010111111100000000) {
        T[eax][0:32] = T[eax][0:32];}
else if (ecx == 0b01010011110100100110010001000001
    ...
    || ecx == 0b00010110001011011011100111100001) {
        T[eax][1:32] = T[eax][0:31];}
...
else T[eax][0:32] = 0;
```

Code 8. Exact mode - TaintInduce (x86) : `shl eax, cl` (Sound)

Memory Indirect Dependencies. Memory indirect accesses are straightforward to handle since there is a direct dependence between index / base registers and the memory address accessed. Consider how TaintInduce recovers memory indirect flows, such as for the `cmovg rax, [bx]` where say the register `rbx` is tainted. Here, our engine generates multiple values of `rbx` and discovers that different memory locations are accessible, i.e. the lower 16 bits of `rbx` has a direct influence on the address. Our algorithm then fixes the concrete value of `rbx` to that found in the trace (`0x1f00`) and fetches the memory content from the simulated memory slot. Further probes to the memory slot reveal the direct influence of the memory value to `rax`. The resulting rule is shown in Code 9. As per our chosen taint propagation policy, we do not propagate taint across memory indirect flows (from `bx` to `rax`) — however, this is merely a policy choice and an analysis could propagate taint for such cases if desired.

```
if ((!ZF&SF&OF) || (!ZF&!SF&!OF)) {
  T[Addr_R1][0:16] = T[ebx][0:16];
  T[eax][0:32]     = T[0x1f00][0:32];
} else {
  T[eax][32:64] = 0;}
```

Code 9. Exact mode - TaintInduce (x86) : `cmovg rax, [bx]` (Sound)

F. Generalization Mode & Completeness

Thus far, we have learned taint rules that observe the behavior of an instruction under certain inputs (or program states), and learn rules that are sound to apply when the instruction evaluates on those inputs. TaintInduce can memoize these rules and apply them each time an instruction evaluates a previously analyzed state. However, in analyzing long sequences of instructions in executions resulting from real-world programs, it is often desirable to *generalize* beyond the previously seen inputs. In the *generalization* mode, TaintInduce carefully trades off soundness for better efficiency. TaintInduce operates in this mode by default.

The key idea is to tune the admissibility of ϕ_j conditions learned for applying an inferred taint rule for output bit j. Specifically, if we relax ϕ_j to include states beyond those observed, then a memoized rule for an instruction can be applied in program states not previously seen. The change to incorporate this generalization is very small. In both modes, observed states are given either `True` or `False` based on their observed behavior. The difference lies in that for exact mode, unseen values are treated as `False` while for generalization mode, we treat them as `Don't-Cares`. Treating unseen values as `False` forces the minimization algorithm to minimize a completely specified boolean function and consider only what has been observed as `True`. On the other hand, `Don't-Cares` allow the minimization algorithm to treat the unseen states as possibly satisfied by the learned ϕ_j. Our generalization strategy is carefully localized to this one change, and it never applies the rules learned for one instruction to be used in another. Of course, future work can explore generalization across classes of instructions.

Example: Generalization helps. We revisit the `shl eax, cl` example to show the generalized version (Code 10) of the sound rule (Code 8). We show an excerpt of the generalized taint rule for 3 values of `cl`. It is easy to see that generalization helps cover more cases than exact mode. In this case, the rule is also sound.

Another example where the generalized rule is sound and helps cover more cases is the `and` example introduced in Section II-B. As we have previously seen, the condition guards

[3] flips in i change output bit j only in a strict subset of observed states

for this instruction only summarizes the observed samples to preserve soundness. As there is no ambiguity in the observed states, the generalized version of this rule encodes the fact that taint is propagated regardless of the program state (Code 5).

```
// flags and taint zeroing are not included for
// clarity; (ecx & 31) is the 5 LSB
val = ecx & 31;
if (val==0)  {T[eax][0:32] = T[eax][0:32];}
if (val==1)  {T[eax][1:32] = T[eax][0:31];}
...
if (val==31) {T[eax][31] = T[eax][0];}
```

Code 10. Generalization mode - TaintInduce (x86) : `shl eax, cl` (Complete and Sound)

Completeness & Soundness Tradeoff. Code 10 is an example where generalization does *not* come at the expense of being unsound. In fact, the learned rules in generalization mode happen to be sound and complete for that example. One cannot hope that TaintInduce achieves completeness provably, since fuzzing all possible input values of instructions is intractable. This is neither the goal of our system nor is claimed here. One might, however, hope that for instruction sets, the inferred rules get close to the complete semantics through the right generalization strategy. Our empirical evaluation shows how often this happens.

IV. Implementation

TaintInduce takes a program, a concrete input, and set of taint source/sinks. Our prototype implementation of TaintInduce, like many other taint engines [32], [42], has two phases. In the first phase, it records the dynamic execution trace of the program under the given inputs. The addresses of values of memory locations accessed in the instruction as well as the complete register state are recorded before and after each instruction in the trace. A number of off-the-shelf tools can be used for this purpose [9], [33], [59]. Our trace collection is implemented using the Intel Pin tool [33], which supports both Windows and Linux user-level applications. This trace is then subject to offline taint analysis in the second phase.

The second phase learns a set of taint rules and applies them to the trace. Both the exact and generalization mode learn rules over the same instructions that they are applied on. The only difference is on the boolean minimization strategy and memoized application of rules in the latter outside of states being trained on. For each instruction in the execution trace, the learning phase starts with the concrete program input states observed augmented with randomly generated seed values. For generating seeds, starting with initial states, we generate more states to observe using simple strategies include bitwalks, bitfills, and setting to min/max values of integer and floating point representations [4].

TaintInduce has a concrete evaluator component to implement this mutate-and-observe functionality. The implementation of the mutation for each observed value in each instruction depends on the choice of the target platform. If the taint analysis is being implemented in an emulator, the values of registers / memory can be observed and mutated directly in the emulator. On real hardware, one could use standard hardware-assisted debugging interfaces such as JTAG [57] or a software debugger. Debuggers commonly provide the facility of reading and writing register / memory and manipulating the program counter. One caveat is dealing with unobservable changes that are outside the view of the chosen target platform. For instance, a system call may modify state that is not visible to a user-level debugger; similarly, an architecture emulator may not expose modification of persistent disk storage. Such instructions can lead TaintInduce to learn "inaccurate" rules primarily because the chosen implementation strategy does not faithfully satisfy the assumptions — it is incomplete in making observations. The best alternative, thus, is to implement TaintInduce as close to real hardware as possible.

For our current prototype, we choose the widely used `unicorn` [18] engine using the `Qemu` [6] emulator as our target platform for concrete evaluation. We choose `Qemu` primarily because of convenience in validating our ideas across multiple architectures. It supports emulations for several widely used architectures that we experimentally report on. The concrete evaluator implementation is straight-forward and closely mirrors the interface outlined in Figure 1 (Section II). Our implementation of the inference engine of TaintInduce consists of $10K$ lines of Python code. The ESPRESSO algorithm is used to perform boolean minimization as described in Section III-D. We use an off-the-shelf C implementation of ESPRESSO [37] and exported an interface to Python. The learnt rules are applied on the execution trace using a taint propagation component of TaintInduce, which consists of $1.2K$ lines of Python code.

V. Evaluation

We evaluate TaintInduce on the following aspects:

1) **Utility in exploit Diagnosis:** Can TaintInduce detect taint-style vulnerabilities in real programs? Does TaintInduce excessively over- or under-taints?
2) **Coverage and Correctness:** In generalization mode, how many instructions across multiple architectures can TaintInduce automatically propagate taint? How does this compare to existing tools?
3) **Cross-referencing utility:** Is TaintInduce effective as a cross-referencing tool, for finding errors in taint engines, emulators, and ISA developer manuals?
4) **Performance:** What is the average cost of learning an instruction on an unknown architecture, and how much efficiency is gained by memoization?

To evaluate these, we use several benchmarks. We measure coverage of TaintInduce stand-alone by testing it with randomly generated values on $1,527$ instruction types across 8 categories across 4 architectures. Further, we evaluate TaintInduce on 15 real-world programs and 26 known CVEs, both on Windows and Linux, with execution traces with millions of tainted (and untainted) instructions. We directly compare TaintInduce to 3 popular implementations of dynamic taint tracking: `TEMU`, `Triton`, and `libdft` which support the x86 architecture which support the same propagation policy as TaintInduce.

Note that all comparisons for correctness (or soundness) for TaintInduce and other tools are automated; our definition allows testing for concrete witnesses that exhibit an output

[4] More advanced strategies which include observation feedback loops can be implemented as an extension in the future.

TABLE I. Summary of CVEs. **Num** is number of instructions. **RCE** is Remote Code Execution, **S-OF** is Stack Overflow, **I-D** is Integer Division-by-Zero, **I-U** is integer underflow. **FP-D** is Floating-Point Division-by-zero, **HC** is Heap Corruption. * represents CVEs which have indirect data propagation.

CVE	Prog	Type	Num	OS
CA-1999-14	bind	RCE	857915	Linux
CA-1999-14	bind	I-U	866934	Linux
CVE-1999-0009	bind	RCE	239825	Linux
CVE-2001-0013	bind	RCE	216774	Linux
CA-2003-07	sendmail	RCE	82999	Linux
CVE-1999-0131	sendmail	S-OF	920086	Linux
CVE-1999-0206	sendmail	RCE	90918	Linux
CVE-1999-0047*	sendmail	RCE	192953	Linux
CA-2003-12*	sendmail	RCE	200018	Linux
CVE-2001-0653	sendmail	I-U	76049	Linux
CVE-2002-0906	sendmail	RCE	106421	Linux
CVE-1999-0878	wu-ftpd	RCE	168604	Linux
CAN-2003-0466	wu-ftpd	RCE	98976	Linux
CVE-1999-0368	wu-ftpd	RCE	185949	Linux
CVE-2003-0352	rpcss	RCE	45328	WinXP
CVE-2002-0649	mssql	RCE	213584	WinXP
CVE-2002-0649	mssql	RCE	551212	Win2k
CVE-2002-1816	atphttpd	RCE	168119	Linux
CVE-2001-0414	ntpd	RCE	26100	Linux
CVE-2003-0201	smbd	RCE	623815	Linux
CVE-2002-1816	ghttpd	RCE	48398	Linux
CVE-2015-6031	miniupnp	S-OF	358896	Linux
CVE-2016-9112	openjpeg2	I-DIV	614908	Linux
CVE-2013-4788	glibc	S-OF	9725	Linux
CVE-2017-14245	libsndfile	FP-DIV	121700	Linux
CVE-2017-7476	gnulib	HC	367930	Linux

change. Wherever TaintInduce runs in generalization mode, for all our experiments, it has been trained on **100 random seed values**, different from concrete trace inputs, before running the propagation tests — the test and training datasets are completely different. The training in exact mode is, by design, on the specific input values being tested on.

Environment Setup. Our experiments are performed on machines spec'd as follows: 64-bit Ubuntu Server 16.04.3 system, with four 8-core Intel Xeon E5-2630 v3@2.48GHz CPUs and 64G RAM. We used Unicorn Engine (version dated Oct 27, 2017) to build TaintInduce. We compared TaintInduce with `Triton` (version dated Nov 10, 2017), `libdft-3.1415alpha` and `TEMU` (Version 1.0).

A. Utility in Exploit Diagnosis

We aim to measure the practical utility of TaintInduce in the offline analysis of memory corruption exploits. We select 26 vulnerabilities of real-world programs with known CVEs used by prior user-level taint engines [11], [43] and whole-system taint analyzers (c.f. `TEMU`) developed in the `BitBlaze` project [51]. In addition, we include several more recent CVEs across a variety of vulnerabilities like stack buffer overflows, heap corruption, floating-point division errors, and integer divide-by-zero. A summary of the programs, vulnerability types and the number of instructions in their execution traces are reported in Table I. In total, we have a total of 26 execution traces totalling $7,454,136$ instructions.

For each of the vulnerable program and given CVE exploit, the taint source are buffers in which the external input is read. TaintInduce propagates taint using the learnt rules and we aim to check if taint reaches the known vulnerability taint sinks (e.g. corrupting the EIP or stack canary). The key result is that the tool successfully propagates taint to all vulnerable sinks, both in exact and generalization modes, for all sinks that satisfy its taint policy.

For 24 of the 26 traces, TaintInduce detects that the taint propagates from the taint source to only the respective taint sinks. For the remaining 2 traces, the final value written to the sink is derived indirectly from an attacker-controlled value. Our standard taint propagation policy (Section III-A) detects but intentionally does not propagate taint for indirect dependencies [5] ; therefore, any standard tainting engines that follow this policy would not detect these CVEs. For the 24 cases that fall within our taint policy, the key result is that even in generalization mode: (a) TaintInduce does not excessively under-taint as taint reaches the known sinks; and (b) TaintInduce does not excessively over-taint, otherwise it could result in attack detection at the wrong control-transfer locations/sinks. Thus, TaintInduce has practical efficacy in the analysis of real-world vulnerabilities. In the remaining 2 cases of indirect flows, TaintInduce propagates taint correctly to attacker-controlled values that indirectly influences the sink.

> *Result 1*: TaintInduce **does not under-taint or over-taint in traces over 7 million instructions on 26 known CVE traces, to propagate taint to (and only to) sinks admissible by its propagation policy.**

B. Coverage on Multiple Architectures

We evaluate TaintInduce on 4 widely-used CPU architectures supported by `unicorn`: *x86*, *x64*, *AArch64* and *MIPS-I*. We obtain a list of instructions from the official developer manuals [1], [3]–[5], [17], [22]. For each operation code, we generate instructions with different operand type combinations. For the generalization mode, we measure the accuracy of each learnt rule on 1000 random test values. To test the correctness of each test, we automatically check if there exists a witness pair of input values which differ in a single bit causing a change in an output bit. This follows directly from our definition of soundness.

In exact mode, TaintInduce learns the sound and complete rule for the values it is tested on by design. Therefore, our remaining results focus on the efficacy of generalization mode on a set of 1000 new random inputs that are used as the test set for the rules. Table II details the number of instruction opcodes for which the learnt rules worked perfectly on all 1000 tests on all operand combinations. It further reports on the number of instruction opcodes supported on `unicorn` and the total number of instructions as per the manual description.

TaintInduce generated sound taint rules for 74.8%, 74.6%, 50.8%, 92.3% of the instructions on x86, x64, ARM

[5] One case is an indirect dependency of a tainted pointer, the taint of which is not propagated to the destination register. Another case is where the taint of the value being conditioned on (`eflags` register) does not propagate to the destination register. Section III-E explains how these cases are handled via conditions.

AArch64, and MIPS-I. For instance, on x86, out of 549 instructions emulated correctly on `unicorn`, TaintInduce learns a sound taint rule for the tested inputs of 411 instructions. For jump, conditional, and data movement, TaintInduce extracted the dependency observations accurately for all of the executable instructions (floating point instructions were recovered accurately). We notice that instructions with an arithmetic component are generally harder to learn for all architectures due to the larger number of states required to observe all possible behaviors. This explains the sharp drop in soundness between ARM64 and x64, i.e., a larger proportion of SIMD instructions for ARM64 have an arithmetic component.

Result 2: TaintInduce **propagates taint information without architecture semantics with** 100% **in exact mode. In *generalization* mode,** TaintInduce **learns sound rules tested over** 1000 **random samples for** 71.51% **of over a thousand of instructions tested across 8 categories in 4 mainstream architectures.**

Examples: Complex Indirect Dependencies. A number of instruction classes have complex conditional dependencies. For instance, conditional instructions like `cmova` on `x86` and `x64`, TaintInduce learns the necessary ϕ conditions to propagate taint soundly. As another example, TaintInduce accurately learns the conditional dependencies in the floating point (FPU) instructions on the `x86` family. The FPU, better known as the `x87` coprocessor, has 8 registers, `st0` to `st7`, which forms a register stack. These registers alias with another set of registers named `fp0-fp7`, and the mapping between the two is controlled by a 3-bit field in the Floating-Point Status Word (FPSW) register called `TOP`. Therefore, the behavior of instructions accessing values via `st0-st7` is conditioned on the `TOP` field values. TaintInduce captures such dependencies automatically and correctly. As an example, Code 11 shows the rule for the instruction `fcmovb st0, st3` which is generated by TaintInduce. The rule highlights the dependence on the floating point register defined in the `TOP` field in the `FPSW` register and `CF` in `EFLAGS`.

```
if (CF) {
  if (TOP == 0) T[fp0] = T[fp3];
  ...
  if (TOP == 7) T[fp7] = T[fp2];}
```

Code 11. TaintInduce (x86) : `fcmovb st(0), st(3)`

Similarly, TaintInduce correctly captures the conditional dependence between the instruction pointer (`eip`) and control-flow instructions in the `CALL` instruction. TaintInduce captures indirect dependencies between register operands that control memory accesses as well. For instance, on `x86`, the `call [eax]` instruction contains implicit operands (`esp` and `eip`) and several direct data dependencies, which TaintInduce accurately learns. TaintInduce learns that the stack space storing the saved `eip` is dependent on `eip`, and `eip` is dependent on the memory content stored at `[eax]`.

Soundness Tradeoff in Generalization mode. In 28.49% instructions, TaintInduce is incorrect on one or more of the 1000 input contexts we tested (but only in generalization mode). One example is the x86 instruction `maxpd xmm1, xmm2` which performs a SIMD compare of the packed double-precision floating-point values in the destination operand (first operand) and the source operand (second operand), and returns the maximum value for each pair of values to the destination operand. The value of the destination operand is determined by an internal computation result ((`xmm1` $\geq$ `xmm2`) == True), rather than a condition from the input. TaintInduce misses the specific condition, and learns an approximate relationship ((xmm1, xmm2)→xmm1) only. Another example where TaintInduce generalizes unsoundly is the `add eax, ebx` instruction (Code 12). The unsoundness stems from the limited sampled states which are used to infer ϕ. Recall that although the rule is unsound in the general sense, it is correct for the set of 100 states it is trained on.

```
if (ebx[6] & !ebx[11] & ebx[17] & !ebx[21])
  T[eax][2] = T[eax][1];
```

Code 12. Incorrect generalization for add eax, ebx

C. Correctness Comparison To Tools

We also compare TaintInduce directly to 3 popular and mature binary-level tainting tools: `TEMU`, `Triton`, and `libdft` for traces over a million instructions. For comparison with `TEMU`, we use the benchmark programs `rpcss`, `mssql`, `atphttpd`, `ntpd`, `smbd`, `ghttpd` presented in Section V-A on the architectures supported (i.e. `x86`). Since these benchmarks do not directly work with `Triton` and `libdft`, we use a second benchmark for testing these two tools on all four architectures listed in Table IV. It consists of 10 programs from the LAVA-M [21] benchmarks, `libtiff` and `binutils` packages used in fuzz testing evaluations [7], [39]. These programs do *not* necessarily have taint-style memory errors in our benchmarks, but we select these because they take tainted file inputs that are extensively processed by the application.

For comparison with a tool, we analyze the taint propagated for each instruction in the designated program execution. To minimize cascading effect due to errors, if TaintInduce and the compared tool differ in output, we record this discrepancy and set the latter's taint output as the taint status for the next instruction — this localizes the checking of taint rules for each instruction, ensuring that discrepancy does not propagate to the next instruction's checking. The comparison procedure is automated. In the event of a discrepancy, we resort to manual analysis against the instruction set manual and CPU behavior.

In comparing with existing tools, we use only the generalization mode in TaintInduce, since exact mode produces strictly superior results to TaintInduce's generalization mode. As in previous setups, TaintInduce is trained on 100 random seed inputs for each instruction occurring in the tested traces. Table III summarizes the coverage of each tool with TaintInduce in generalization mode. TaintInduce has less than 7% discrepancies from these mature tools with hand-crafted rules, and only in 0.28% of these cases is TaintInduce incorrect. The test is done automatically using witness values with bit-flips on the real CPU.

Result 3: TaintInduce **learns rules that propagate identically to existing tools between** 93.27% **and** 99.5%**, without requiring any architectural semantics. Only** 0.28% **of the discrepancies are errors in** TaintInduce**, the rest are errors in state-of-the-art implementations.**

TABLE II. ARCHITECTURE SUPPORT OF TAINTINDUCE. **TOTAL (T)** IS THE TOTAL NUMBER OF EXECUTABLE INSTRUCTIONS ON `UNICORN`, **SUPPORT (P)** IS THE TOTAL NUMBER OF INSTRUCTIONS FOR WHICH TAINTINDUCE GENERATES RULES WITHOUT INPUT SENSITIVITY, **SOUND (S)** IS THE TOTAL NUMBER OF INSTRUCTION FOR WHICH TAINTINDUCE GENERATES SOUND RULES IN GENERALIZED MODE.

Architecture	Type of Instructions																								Instruction set		
	Arith			Comp			Jump			Mov			Cond			FPU			SIMD			MISC			Sound	Support	Total
	S	P	T	S	P	T	S	P	T	S	P	T	S	P	T	S	P	T	S	P	T	S	P	T			
x86	28	43	43	0	8	8	33	33	33	32	32	32	60	60	60	59	83	86	176	251	259	23	26	28	411	536	549
x64	23	38	38	0	10	10	35	35	35	39	39	39	60	60	60	59	83	86	176	251	259	23	26	29	415	542	556
Aarch64	39	64	64	0	3	3	3	3	3	43	43	43	11	11	11	18	37	37	61	188	196	13	13	13	188	362	370
MIPS-I	22	26	26	4	4	4	7	7	7	14	14	14	-			-			-			1	1	1	48	52	52

TABLE III. COVERAGE OF TAINTINDUCE, `LIBDFT`,`TRITON` AND TEMU ON `X86`. × MEANS UNSUPPORTED. √ MEANS SUPPORTED.

Tool	Type of Instructions(Support)									Taint level	Register Support			
	Arith	Comp	Jump	Mov	Cond	FPU	SIMD	MISC	Total		Purpose	EFLAGS	FPU	XMM
TAINTINDUCE	43	8	33	32	60	83	251	26	536	BIT	√	√	√	√
libdft	15	5	1	27	33	×	×	8	89	BYTE	8 Basic	×	×	×
Triton	38	7	19	32	33	×	127	9	265	REG	√	√	√	√
TEMU	7	1	2	1	×	×	×	×	11	BYTE	√	√	×	×

TABLE IV. COMPARISON OF TAINTINDUCE WITH `LIBDFT` AND `TRITON`. WE COUNT THE TOTAL NUMBER OF INSTRUCTIONS (`TRACE TOTAL`), THE NUMBER OF UNIQUE INSTRUCTIONS (`UNIQUE`) AND THE NUMBER OF INSTRUCTIONS THAT HAVE AT LEAST ONE TAINTED OPERAND (`TAINTED`) FOR EACH BINARY. WE MEASURE THE MISMATCH WITH `LIBDFT` AND `TRITON` WITH THE TOTAL NUMBER OF INSTRUCTIONS (T) AND THE NUMBER OF UNIQUE INSTRUCTIONS (U). FOR THESE MISMATCHES, WE SHOW HOW MANY ARE DUE TO WRONGLY IMPLEMENTED RULES (`IMPL RULES`), INSUFFICIENT SUPPORT FOR INSTRUCTIONS (`INS SUPP`) FOR `LIBDFT`, INSUFFICIENT GRANULARITY FOR `TRITON` (`INS GR`), INPUT CONTEXT INSENSITIVITY FOR `LIBDFT`, `TRITON` (`GEN RULES`) AND INSUFFICIENT OBSERVATIONS IN TAINTINDUCE (`TI INS SAMPLES`).

Binary	libdft											Triton												
	Trace Total	Unique	Tainted	Mismatch		Reason for Mismatch						Trace Total	Unique	Tainted	Mismatch		Reason for Mismatch							
						Impl Rules		Ins Supp		Gen Rules							Impl Rules		Ins Gr		TI Ins Samples		Gen Rules	
				T	U	T	U	T	U	T	U				T	U	T	U	T	U	T	U	T	U
base64	283132	6244	42071	10660	33	6110	16	4482	9	68	8	281159	6182	42194	11267	150	149	5	10688	37	281	93	149	15
who	2031615	19175	549350	201757	94	114592	79	85842	12	1323	3	320765	6699	45284	13908	158	212	8	13380	82	213	61	103	7
uniq	2097151	6242	932802	69476	56	63889	36	4270	17	1317	3	326395	5527	9655	388	6	0	0	388	6	0	0	0	0
md5sum	2670592	7712	11886	482	69	295	40	135	17	52	12	310780	6042	375	94	7	45	2	25	3	24	2	0	0
tiffsplit	655359	6434	339112	749	51	710	35	18	8	21	8	515325	5888	110503	859	125	416	15	280	70	147	35	16	5
tiff2pdf	1048575	8854	648397	2771	65	2632	46	106	12	33	7	550396	5377	120306	867	87	545	9	182	42	139	35	1	1
tiff2rgba	2684353	6697	2329387	96191	88	80687	67	15300	13	204	8	523517	6223	78515	1688	147	300	16	767	86	593	38	28	7
bmp2tiff	2162687	6061	1774856	103814	29	103762	15	17	7	35	7	527613	5259	79874	4790	50	52	11	4709	26	15	6	14	7
objdump	2682463	10568	677219	12255	116	8593	79	3325	25	337	12	433370	5539	18614	426	40	120	8	222	23	84	9	0	0
readelf	1106687	8413	233400	9060	60	6480	38	2506	15	74	7	324622	6249	1257	2	1	0	0	2	1	0	0	0	0
Total	**17422614**	**31050**	**7538480**	**507215**	**661**	**387750**	**451**	**116001**	**135**	**3464**	**75**	**4113942**	**12987**	**506577**	**34289**	**771**	**1839**	**74**	**30643**	**376**	**1496**	**279**	**311**	**42**

Comparison with `TEMU`. `TEMU` keeps a taint record for inputs of each instruction. Of the total $1,676,556$ instructions in the benchmark traces, TAINTINDUCE produces the same taint output in 99.5% of the times. In the cases where TAINTINDUCE and `TEMU` disagree, we find that the taint semantics TAINTINDUCE generated are correct. Our manual analysis on the unique instructions confirms that these errors in `TEMU` both over- and under-taints due to implementation bugs.

Comparison to `libdft` and `Triton`. We compare these tools against TAINTINDUCE for each instruction in the second benchmark. There is a total of $21,536,556$ instructions in all traces generated by both tools, out of which $31,050$ are unique. Of the $21,536,556$ instructions, a total of $8,045,057$ instructions are tainted instructions that have taint propagated or cleared by `Triton` or `libdft`. First, for 93.27% of these tainted instructions, TAINTINDUCE output agrees with the compared tool, showing that our approach performs well without knowledge of specialized rules. Since we are doing bit-level tracking, for comparison, we consider the byte/register tainted if all bits are reported as tainted by TAINTINDUCE.

On the remaining $541,504$ instructions executed, TAINTINDUCE disagrees with either `libDFT` or `Triton`. 21.42% of the discrepancies are due to unsupported instructions in `libdft`, namely `shl`, `shr`, `movd`, `shld`, `fst` and `pmovmskb`. For these instructions, `libdft` silently performs a `nop` for the taint propagation. Another 78.3% of the discrepancies are because the compared tools approximate by tracking at the level of byte or register level, the implemented rules are incorrect or often ignore conditional behavior, applying the same rule across all input contexts. All these missed nuances are important in the concretely tested executions. An example which highlights these issues is the `movzx` instruction which zero extends the value to 16 or 32 bits. Specifically, for `movzx eax, bx` where `bx` is tainted, the correct influence will be that the lower 16 bits of `ebx` propagate into the lower 16 bits of `eax` and the taint of the upper 16 bits of `eax` are cleared. While in this case, byte-level granularity is sufficient to accurately propagate the taint information, with the lower two bytes tainted and top two bytes untainted, `libdft` unsoundly sets the taint for the entire 32-bit register because of an incorrectly implemented taint rule. `Triton` works on a register-level granularity so we mark this mismatch as under the category of "insufficient granularity" in Table IV.

Finally, only 0.28% of the discrepancies are because TAINTINDUCE infers the wrong rule. All of the cases missed by TAINTINDUCE correspond to arithmetic and logical operations with immediate values where TAINTINDUCE missed the condition where the `ZF` flag register bit is set on x86/x64. No values in our tests exhibit this behavior during learning, and TAINTINDUCE memoizes a rule that under-taints the `ZF` flag.

All engines differ. On the x86 architecture, logical instructions such as `and` perform bit-wise operations on the destination and source operands, and store the results in the destination operand locations. Consider the `and` instruction with an immediate as the source operand, specifically `and eax, 0x16`. Both `libdft` and `Triton` simply preserve the taint of the destination operand. TEMU performs an additional check if the immediate value is 0, and if so, it clears the taint of the destination operand instead. In this case, all three taint engines wrongly propagate taint to the destination operand. On the other hand, TaintInduce identifies that only the taint of the 4^{th} bit of eax should be preserved, while the rest should be cleared. Also, the taint computation for flags is incorrect for all 3 engines. libdft does not perform taint tracking on the flags, Triton simply clears the taint for OF and CF and propagates taint to PF, SF, and ZF. Although TEMU checks for a specific case (the immediate 0), it propagates the taint to all 5 flags no matter the value of the immediate. PF and SF are always set to zero and since the inputs have no influence over PF and SF, the taint should be cleared instead — only TaintInduce learns the correct rule.

D. Utility as a Cross-Referencing Tool

During our correctness testing, we resolve discrepancies between TaintInduce and the other tools by consulting the instruction set manuals and documentation [17]. We find a set of cases where the instruction set specification documentation is either wrong (inconsistent with the CPU implementation), ambiguous or left to CPU implementation choices intentionally. These leads to taint propagation errors, highlighting the subtleties in writing rules and that ISA manuals are not reliable as the source of ground truth. Furthermore, several cases in which TaintInduce had propagation errors are due to its concrete observation sub-tool, namely unicorn. In total, we found 11 rule errors in libdft, 7 rule errors in Triton, 2 rule errors in TEMU, 17 emulation errors in unicorn, and 3 descriptions in instruction manuals.

Result 4: **When cross-referencing with TaintInduce, we find 20 bugs in existing taint tools, 17 errors in unicorn, 3 description errors (or ambiguity) in ISA instruction manuals.**

ISA Manual Errors. The Intel's Software Developers Manual specifies the behavior of x86 instructions over 2000 pages. We find that TaintInduce reports a taint discrepancy that does not match the description of bt r16/r32, r16/r32 the Intel manual [17]. Upon concrete testing on a real CPU, we confirm that the documentation is incorrect.

The bt r16/r32, r16/r32 instruction returns a bit located at bit offset specified by the second operand, in a bit string specified by the first operand (called the *bit base*); the result is in the CF flag register bit. TaintInduce identified that the lower order bits 4 or 5 of the bit offset operand will affect the CF flag, while the manual[6] states that it is the lower order bits 3 or 5. The correct semantics for the instruction should be 4 bits for 16-bit operand and 5 bits for 32-bit operand. This highlights how TaintInduce can be useful to identify description errors in manuals.

Ambiguous Specifications in Manuals. There is intentional ambiguity at times in the documentation of instructions. One example we encountered is the bsf instruction [17]. The Intel manual left the behavior of the destination operand undefined. This means that the behavior of the instruction is dependent on the particular implementation of the CPU/emulator. On the other hand, AMD's manual [52] specifies that the destination operand is unchanged when the source operand is zero.

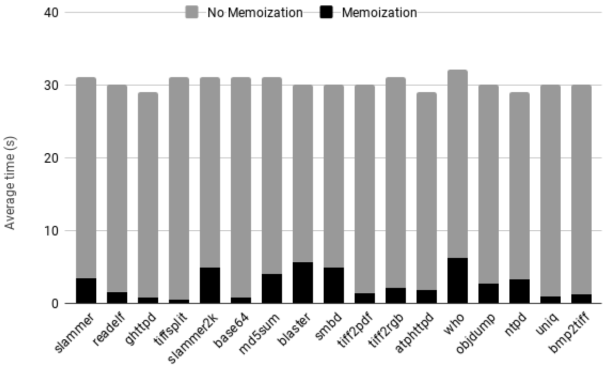


Fig. 3. Average time to train each instruction on one machine. Memoized timings are obtained by training on all other programs before the target.

CPU Implementation Differences. As a final example, we present a case that highlights differences in two CPU implementations of an ISA. The tzcnt is an instruction which counts the number of trailing zeros and is an extension of the bsf instruction. The key difference between tzcnt and bsf instruction is that tzcnt provides operand size as output when source operand is zero while for the bsf instruction, if source operand is zero, the content of destination operand is undefined. On CPUs that do not support the tzcnt instruction, the instruction bytecode will instead be interpreted as a bsf instruction and executed as such. Since our tool infers the behavior through concrete execution, it correctly captures the behavior of the emulator.

Missing emulation in unicorn Engine. When cross-referencing TaintInduce with other engines, we find several errors due to bugs and missing support in our concrete evaluator sub-component which is off-the-shelf unicorn engine. On x86 and x64 architecture, unicorn does not emulate most of SSE4 instructions correctly; does not implement the mask registers; and does not support system and memory cache instruction without execution context. Similarly on AArch64, instructions such as cbz, system instructions (yield, wfe, wfi, sev, sevl), jump instructions and mrs are not supported because unicorn cannot provide the running context they need and does not define the exception link register. As such, for these instructions, we are unable to obtain the observations needed for TaintInduce to infer the rules. For MIPS-I instructions, such as the arithmetic instructions (mult, multu, div) and movement instructions (mfi, mflo, mthi, mtlo), they use the hi or low as its operands. Through our analysis, we find that hi and lo registers are not implemented in unicorn.

E. Performance

The predominant use of taint tracking is in offline analyses. TaintInduce can be run once per architecture offline, and the learned rules are memoized and used for a large number of programs on that architecture. The approach taken in TaintInduce is embarrassingly parallel. In our experiments,

[6] Intel Software Developers Manual, Vol. 2A 3-113

we find that the the average time to learn an instruction without memoization is about 30 seconds on 1 machine. TaintInduce can memoize rules which reduces the average time required to learn unique instructions in our benchmarks to $1 - 7$ seconds, which is a factor of $4x - 30x$ reduction, as shown in Figure 3. Such memoization is effective because the number of unique instructions in our benchmark execution $(44,171)$ is 3 orders of magnitude lesser than the total number of instructions. $13,764$ of the unique instructions are shared by at least two programs. Recall that two instructions with the same opcode but different immediate values are treated as two separate instructions (as they have different encodings); $24,738$ out of $44,171$ instructions are with immediates. TaintInduce does not generalize across instructions with the same opcode though it is a promising direction for future work. For all 27 traces using 20 machines, rule inference took 23 hours while taint propagation took 30 minutes.

Result 5: **Average time to learn an instruction on 1 machine in our benchmarks is 30 seconds, and improves by $4 - 30x$ due to memoization.**

VI. Related Work

There has been more than a decade of research into the deductive approach to taint propagation [10], [20], [29], [32], [42], [51]. The strengths and pitfalls of taint propagation policies on benign-but-buggy software and malware are well-known [12], [47], [50]. TaintInduce does not change the status quo on the efficacy (FPs vs. FNs) of taint policies. However, in all these works, taint rules are manually specified. In contrast, we take an inductive approach of inferring taint rules that adhere to a chosen policy.

Inferring Taint. Some works have explored the idea of propagating taint information through inference rather than manual specification of rules [34], [48]. Both approaches proposed the usage of observation between input and output to infer taint, Sekar [48] for web-based attacks and Matthias et. al. for Android applications [34]. TaintInduce is the first work to target general-purpose computation, such as that of complex instruction sets. The rules learnt are composable across instructions, and we show how to handle complex bit-level taint propagation policies comparable to those used in complementary deductive approaches.

Instruction Semantics Inference. There have been various efforts to automate the creation of semantic definitions of instructions [25], [27], [28], [30]. These prior works make heavy use of SMT solvers and templates derived from domain knowledge like program sketches that encode simple semantics [25]. While these show that recovering the full semantics of instructions is a hard problem requiring intimate knowledge about the architecture, we present a technique for recovering influence semantics that is feasible in a blackbox setting.

Soundness of Taint. While a large body of work has been concerned with relating security properties to information-flow control policies [15], [26], [41], only recently a soundness criterion has been proposed specifically for taint tracking [45]. This and other traditional soundness reasoning frameworks on information flow are defined with respect to some operational semantics [47]. DECAF [29] for example, defines taint rules and encodes instruction semantics into SMT theories to guarantee completeness and soundness of its taint rules for integer arithmetic. However, in our problem setup we do not have access to the operational semantics; hence we require a different soundness definition closer to that used for symbolic execution by Godefroid [24]. McCamant et al. [35] propose a soundness definition using entropy based on information flow. TaintInduce uses an existential influence observation rather than a quantitative notion.

VII. Conclusion

In this paper, we present a novel approach that automatically infers taint propagation rules in an architecture-agnostic manner. Our evaluation shows how TaintInduce learns rules for x86, x64, ARM, and MIPS instruction sets. It performs comparably to 3 popular taint tools and supports more instructions making it useful as both a stand-alone taint tool or as a complement to existing taint tools. TaintInduce is also able to detect a range of vulnerabilities for 24 CVEs across both Linux and Windows applications. Furthermore, TaintInduce can also be used to identify implementation bugs in taint engines, emulators or ISA documentations. More information on the tool and webservice can be found on the project page at https://taintinduce.github.io/.

Acknowledgments

We thank the anonymous reviewers of this work for their helpful feedback. We thank R. Sekar, Chia Yuan Cho, Wei Ming Koo, Adrian Tang and Anselm Nicholas for their valuable feedback on earlier drafts of this paper. We also thank Kaihang Ji and Vinamra Bhatia for their help with implementation and experiments. This research is supported in part by the National Research Foundation, Prime Ministers Office, Singapore under its National Cybersecurity R&D Program (Award No. NRF2014NCR-NCR001-21), in part by Grant DSOCL17019 from DSO, Singapore, in part by National Natural Science Foundation of China (Grant No. U1736209, 61572483 and 61502469). All opinions expressed in this paper are solely those of the authors.

References

[1] "https://en.wikipedia.prg/wiki/X86_instruction_listings," 2017.

[2] "https://software.intel.com/en-us/blogs/2017/10/17/does-software-actually-use-new-instruction-sets," 2017.

[3] "https://www.c9x.me/x86/," 2017.

[4] "http://www.felixcloutier.com/x86/index.xml," 2017.

[5] ARM Limited, "ARMv8 Instruction Set Overview," pp. 1–115, 2013.

[6] F. Bellard, "QEMU , a Fast and Portable Dynamic Translator," *ATC*, pp. 41–46, 2005.

[7] M. Böhme, V.-T. Pham, and A. Roychoudhury, "Coverage-based greybox fuzzing as markov chain," in *CCS*. ACM, 2016, pp. 1032–1043.

[8] R. K. Brayton, G. D. Hachtel, C. McMullen, and A. Sangiovanni-Vincentelli, *Logic minimization algorithms for VLSI synthesis*. Springer Science & Business Media, 1984, vol. 2.

[9] D. Bruening, Q. Zhao, and S. Amarasinghe, "Transparent dynamic instrumentation," *ACM SIGPLAN Notices*, vol. 47, no. 7, pp. 133–144, 2012.

[10] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A Binary Analysis Platform," in *CAV*, 2011.

[11] J. Caballero, Z. Liang, P. Poosankam, and D. Song, "Towards generating high coverage vulnerability-based signatures with protocol-level constraint-guided exploration," in *RAID*. Springer, 2009, pp. 161–181.

[12] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *DIMVA*. Springer, 2008, pp. 143–163.

[13] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding Data Lifetime via Whole System Simulation," in *USENIX Security*, 2004.

[14] J. Clause, W. Li, and A. Orso, "Dytan: A Generic Dynamic Taint Analysis Framework," in *ISSTA*. New York, New York, USA: ACM Press, 2007.

[15] E. S. Cohen, "Information transmission in sequential programs," *Foundations of Secure Computation*, pp. 297–335, 1978.

[16] P. M. Comparetti, G. Wondracek, C. Kruegel, and E. Kirda, "Prospex: Protocol specification extraction," in *S&P*, 2009.

[17] I. Corporation, "Intel ® 64 and IA-32 Architectures Software Developer's Manual," vol. Volume 2, 2011.

[18] H.-V. Dang and A.-Q. Nguyen, "Unicorn: Next generation cpu emulator framework," 01 2015.

[19] D. E. Denning and P. J. Denning, "Certification of programs for secure information flow," *Communications of the ACM*, vol. 20, no. 7, pp. 504–513, 1977.

[20] B. Dolan-Gavitt, J. Hodosh, P. Hulin, T. Leek, and R. Whelan, "Repeatable Reverse Engineering with PANDA," in *PPREW*, 2015, pp. 1–11.

[21] B. Dolan-Gavitt, P. Hulin, E. Kirda, T. Leek, A. Mambretti, W. Robertson, F. Ulrich, and R. Whelan, "Lava: Large-scale automated vulnerability addition," in *S&P*. IEEE, 2016, pp. 110–121.

[22] D. J. Ellard, "MIPS Assembly Language Programming CS50 Discussion and Project Book," 1994.

[23] V. Ganesh, T. Leek, and M. Rinard, "Taint-based directed whitebox fuzzing," in *ICSE*, 2009.

[24] P. Godefroid, "Higher-order test generation," in *ACM SIGPLAN Notices*, vol. 46, no. 6. ACM, 2011, pp. 258–269.

[25] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from I/O samples," in *PLDI*, 2012.

[26] J. A. Goguen and J. Meseguer, "Security policies and security models," in *S&P*. IEEE, 1982, pp. 11–11.

[27] N. Hasabnis and R. Sekar, "Extracting instruction semantics via symbolic execution of code generators," in *FSE*, 2016.

[28] ——, "Lifting assembly to intermediate representation: A novel approach leveraging compilers," *ACM SIGPLAN Notices*, vol. 51, no. 4, pp. 311–324, 2016.

[29] A. Henderson, A. Prakash, L. K. Yan, X. Hu, X. Wang, R. Zhou, and H. Yin, "Make it work, make it right, make it fast: building a platform-neutral whole-system dynamic binary analysis platform," in *ISSTA*, 2014.

[30] S. Heule, E. Schkufza, R. Sharma, and A. Aiken, "Stratified synthesis: automatically learning the x86-64 instruction set," in *ACM SIGPLAN Notices*, vol. 51, no. 6. ACM, 2016, pp. 237–250.

[31] M. G. Kang, S. Mccamant, and P. Poosankam, "DTA++: Dynamic Taint Analysis with Targeted Control-Flow Propagation," in *NDSS*, 2011.

[32] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "Libdft: Practical Dynamic Data Flow Tracking for Commodity Systems," in *VEE*, 2012.

[33] 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 *ACM SIGPLAN Notices*, vol. 40, no. 6. ACM, 2005, pp. 190–200.

[34] B. Mathis, V. Avdiienko, E. O. Soremekun, M. Böhme, and A. Zeller, "Detecting information flow by mutating input data," in *ASE*. IEEE Press, 2017, pp. 263–273.

[35] S. McCamant and M. D. Ernst, "Quantitative information flow as network flow capacity," in *ACM SIGPLAN Notices*, vol. 43, no. 6. ACM, 2008, pp. 193–205.

[36] E. J. McCluskey, "Minimization of boolean functions," *Bell Labs Technical Journal*, vol. 35, no. 6, pp. 1417–1444, 1956.

[37] P. McGeer, J. Sanghavi, R. Brayton, and A. S. Vincentelli, "Espresso-signature: A new exact minimizer for logic functions," in *DAC*. ACM, 1993, pp. 618–624.

[38] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *NDSS*, 2005.

[39] S. Rawat, V. Jain, A. Kumar, L. Cojocar, C. Giuffrida, and H. Bos, "Vuzzer: Application-aware evolutionary fuzzing," 2017.

[40] K. A. Roundy and B. P. Miller, "Binary-code obfuscations in prevalent packer tools," *ACM Computing Surveys (CSUR)*, vol. 46, no. 1, p. 4, 2013.

[41] A. Sabelfeld and A. C. Myers, "Language-based information-flow security," *IEEE Journal on selected areas in communications*, vol. 21, no. 1, pp. 5–19, 2003.

[42] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *SSTIC*. SSTIC, 2015, pp. 31–54.

[43] P. Saxena, P. Poosankam, S. McCamant, and D. Song, "Loop-extended symbolic execution on binary programs," in *ISSTA*. ACM, 2009, pp. 225–236.

[44] P. Saxena, R. Sekar, and V. Puranik, "Efficient fine-grained binary instrumentation with applications to taint-tracking," in *CGO*, 2008.

[45] D. Schoepe, M. Balliu, B. C. Pierce, and A. Sabelfeld, "Explicit secrecy: A policy for taint tracking," in *EuroS&P*. IEEE, 2016, pp. 15–30.

[46] J. Schutte, D. Titze, and J. M. Fuentes, "AppCaulk: Data leak prevention by injecting targeted taint tracking into android apps," in *TrustCom*, 2015.

[47] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All You Ever Wanted to Know About Dynamic Taint Analysis Forward Symbolic Execution (but might have been afraid to ask)," in *S&P*, 2010.

[48] R. Sekar, "An efficient black-box technique for defeating web application attacks." in *NDSS*, 2009.

[49] U. Shankar, K. Talwar, J. S. Foster, and D. Wagner, "Detecting format string vulnerabilities with type qaualifiers," in *USENIX Security*, 2001.

[50] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *CCS*. ACM, 2009, pp. 61–74.

[51] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A New Approach to Computer Security via Binary Analysis," in *Proceedings of the 4th International Conference on Information Systems Security*, 2008.

[52] A. Technology, "Amd64 technology amd64 architecture programmer's manual volume 3: General-purpose and system instructions publication no. revision date," 2012.

[53] O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, O. Weisman, O. Tripp, M. Pistoia, S. J. Fink, M. Sridharan, and O. Weisman, "TAJ: effective taint analysis of web applications," in *PLDI*, 2009.

[54] C. Umans, T. Villa, and A. L. Sangiovanni-Vincentelli, "Complexity of two-level logic minimization," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, no. 7, pp. 1230–1246, 2006.

[55] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross-Site Scripting Prevention with Dynamic Data Tainting and Static Analysis," in *NDSS*, 2007.

[56] T. Wang, T. Wei, G. Gu, and W. Zou, "TaintScope: A checksum-aware directed fuzzing tool for automatic software vulnerability detection," in *S&P*, 2010.

[57] Wikipedia contributors, "Jtag — Wikipedia, the free encyclopedia," https://en.wikipedia.org/w/index.php?title=JTAG&oldid=851233690, 2018, [Online; accessed 7-August-2018].

[58] W. Xu, S. Bhatkar, and R. Sekar, "Taint-enhanced policy enforcement: A practical approach to defeat a wide range of attacks." in *USENIX Security*, 2006, pp. 121–136.

[59] H. Yin and D. Song, "Temu: Binary code analysis via whole-system layered annotative execution," *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2010-3*, 2010.

[60] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing System-wide Information Flow for Malware Detection and Analysis," in *CCS*. New York, New York, USA: ACM Press, 2007.

[61] 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*, vol. 45, no. 1, pp. 142–154, 2011.