SoK: On the Soundness and Precision of Dynamic Taint Analysis

Lok Kwong Yan[†‡] Heng Yin[†]

[†]*Syracuse University*
Syracuse, New York, USA

[‡]*Air Force Research Laboratory*
Rome, New York, USA

{*loyan, heyin*}*@syr.edu*

***Abstract*—Taint analysis or dynamic information flow tracking is a key binary analysis technique for revealing data dependencies in programs. It has been used in many different applications, such as memory error detection, vulnerability analysis, malware analysis, and exploit diagnosis. While previous implementations are empirically effective for their chosen tasks, they use manually defined tainting rules which have not been proven to be sound (lack false negatives) or precise (lack false positives). Furthermore, even perfect tainting rules could be incorrectly implemented. We survey a number of existing systems, and find that all suffer from either unsoundness or imprecision, and many are quite imprecise. To improve the situation, we propose a set of formal methods to create tainting rules and verify their soundness and precision, and to verify the correctness of tainting implementations. To show the practicality of this approach, we build a new taint analysis system that is formally verified to be sound and in many cases precise at the instruction level. Our tests with real world workloads (tainted shell commands in both Windows and Linux, and keylogger detection) demonstrate observable advantages of having provable soundness and precision, as compared to existing taint analysis systems.**

I. Introduction

Dynamic taint analysis [1] (also called tainting, dynamic information flow tracking, etc.) is a fundamental dynamic analysis technique. The key idea is to label certain data values (such as memory and CPU register contents) as tainted and propagate the taints through operands as instructions execute. A taint propagation rule (tainting rule or rule in short) is associated with each instruction (sometimes with special sub-cases), specifying whether each output operand should be tainted or untainted based on the taint status of the input operands.

While the core idea is simple, dynamic taint analysis has been demonstrated to be crucial in solving many security problems, such as exploit detection and analysis [1], malware analysis [2], protocol reverse engineering [3], vulnerability signature generation [4], guided fuzz testing [5], etc. It is also a foundation for mixed concrete and symbolic execution [6].

Because of its importance, there exist many implementations in the literature. Some implementations were built upon process-level dynamic binary translation, including TaintCheck [1], Dytan [7], LIFT [8], libdft [9], and Minemu [10]. Another set of implementations is based on CPU emulators (e.g., QEMU [11] and Bochs [12]), such as TaintBochs [13], Argos [14], TEMU [15], and DroidScope [16]. There are even implementations in hardware [17], [18], [19], [20], [21], [22], [23]. These implementations have different taint granularities (e.g., labeling bytes vs. bits). They also use different tainting rules that are manually specified to suit their own problem domains.

While these previous systems have demonstrated their success on solving various security problems, little effort has been made to formally examine the correctness of their taint analysis implementations. An ideal taint analysis system needs to be *sound* and *precise*. Informally, a taint analysis system is sound if all data items that are supposed to be tainted are marked as tainted by the system, and precise if only the data items marked as tainted by the system are those supposed to be tainted. In other words, a sound implementation has no false negatives, and a precise implementation has no false positives. When dealing with security problems, an unsound implementation may miss real attacks, while an imprecise implementation may raise too many false alarms.

In this paper, we take the first steps towards tackling this fundamental problem in dynamic taint analysis. We propose techniques for verifying the soundness and precision of taint propagation rules that can be applied either at design time or to verify an existing implementation. Our goal is a dynamic taint analysis system which is both sound and precise. Since most of the existing taint analysis implementations operate at the instruction level, we currently focus on providing this correctness guarantee at the instruction level. Soundness at the instruction level also guarantees soundness at higher levels. Extending a precision guarantee to a larger code granularity, such as a basic block or a function body, is left as future work.

Formally verified systems sometimes have a reputation for having limited functionality or being less practical than ad-hoc designs. To show that this is not the case for a taint analysis system, we also implement a practical security-oriented dynamic taint analysis tool as a case study of our approach, and structure the paper around this development process.

Our first step is "tainting rule creation and verification", in which we create tainting rules for different instructions and verify their soundness and precision. A successful first step results in a collection of correct taint propagation rules. To achieve this, we formally model the internal tainting logic of each instruction against definitions based on noninterference,

generate and refine candidate rules, and then use automated decision procedures (Z3 [24] for the quantified and unquantified logic of bitvectors and MONA [25] for weak monadic second-order logic) to verify that the correctness definitions are satisfied. We are able to create a set of rules and verify that all of them are sound and most are precise.

The second step is "tainting rules implementation", in which we implement these rules in a dynamic taint analysis system called SPITA. In particular, we have implemented the set of verified tainting rules as an extension to QEMU, a CPU emulator. QEMU uses a compiler-like backend called TCG (Tiny Code Generator) for dynamic binary translation. Therefore, we have implemented these tainting rules primarily using the TCG IR (intermediate representation). These tainting IRs are inserted into the original IR block to implement the specified tainting rules. This IR-level implementation makes it easier to cover a large number of instructions: for instance a single rule for an IR addition operation can be used in many different x86 instructions that perform addition.

The third step is "tainting system verification", in which we conduct per-trace formal verification with our tool VITA, to make sure the implementation exactly follows the specification (i.e., the tainting rules) in a large collection of test scenarios. We leverage a suite of over 600,000 test programs generated using the PokeEMU [26] system. PokeEMU symbolicly executes the Bochs x86 emulator to achieve high path coverage over a large set of instructions, therefore the tests provide extensive examples of many processor behaviors. For each test case, we collect an execution trace that contains instruction opcodes, concrete values for instruction operands, and tainted values for each operand before and after the instruction is processed. Then, we use a noninterference oracle to verify that the resulting taint labels in the trace match our formal model of correctness.

After going through these three steps, SPITA has now been formally verified to be sound and mostly precise at the instruction level. To demonstrate the practicality of SPITA, we conducted two case studies, showing that improved precision leads to less unnecessary taint in large applications.

Contributions: In short, we have made the following contributions:

- We adopted noninterference to formally model taint propagation correctness in a data-centric style.
- Based on this model, we conducted a survey of publicly available taint analysis systems and found many cases of unsoundness and extensive imprecision.
- Again based on the same model, we created our tainting rules and checked their soundness and precision. Our verification showed that all the rules are sound and most are precise.
- We implemented these tainting rules in our prototype taint analysis system called SPITA, based on QEMU. It currently supports all integer-based instructions on the x86 target. The tainting rules for all of them are sound and most are precise.
- With over 600,000 test cases, we conducted per-trace verification to ensure the correctness of our implementation. Through this verification, we indeed found a few implementation bugs and fixed them.
- We conducted case studies to show that with the guarantee of soundness and more precision, SPITA achieves significantly better results on keystroke tainting. The evaluation of a large set of keyloggers further demonstrated its effectiveness.

II. Formal Model and Definitions

This section starts with an overview of our data-centric noninterference model used to analyze instruction level taint trackers. We also make observations on the model and how it relates to taint tracker implementations in practice. This discussion helps motivate some of our design decisions. Finally, this section concludes with the definitions we use for formal verification of taint propagation rules and taint analysis implementations.

The original formulation of noninterference by Goguen and Meseguer [27] was applied to a multi-level secure operating system and used a state-machine model. A more modern formulation divides the state of an arbitrary system into two parts, named "high" and "low." Consider two possible starting states of the computation that are the same in their low portions ("low-equivalent"), though the high portions may be different. If the computation satisfies noninterference, then the output states of the computation on those two inputs will also be low-equivalent. Intuitively, this definition captures a lack of information flow from high to low.

When we apply the noninterference principle to dynamic taint analysis, the tainted values correspond to high. Noninterference is a soundness property for tainting, saying intuitively that tainted values before the computation never affect untainted values after, or equivalently that any value affected by a tainted value is itself tainted. We also want precise tainting: subject to the constraint of noninterference, the amount of data tainted should be as small as possible.

The usual definition of noninterference considers the entire tainted (high) state of a system, but for reasoning about noninterference it suffices to consider the effect of changing an arbitrarily small part of the state. Stated informally, if a large change has an effect, then among the smaller changes that make it up, at least one must also have an effect. Taking advantage of this property, we narrow our analysis to consider the effect of the smallest possible change: changing a single bit from 0 to 1 or vice versa.

For the purposes of this paper we will model the state of the computation system (e.g., a CPU) as a vector of bits. We use the symbols $\wedge$, $\vee$, $\oplus$, and an overbar to represent the Boolean operations of AND, OR, XOR, and NOT either on single bits or bitvectors, equivalent to the `&`, `|`, `^` and `~` operators in C. S is the set of possible states, equal to all the bitvectors of a particular fixed size. We identify bit positions with bitvectors that have just that one bit set, and use the notation $v|_b$ for extracting a single bit b from a bitvector v.

Definition 1: Let a and b be bits in the state of a system. We say that a computation has an *information flow from a to b* if there are two input states s_0 and s_1 that are identical except that s_0 has $a = 0$ and s_1 has $a = 1$, and in the corresponding output states s'_0 and s'_1, the values of b are different (one 0 and the other 1, in either order). In other words, if the computation is a function f on state vectors, and a is a bitvector with only a single position set to `1`, there is a state vector $s \in S$ such that:

$$f(s \vee a)|_b \neq f(s \wedge \overline{a})|_b \tag{1}$$

From the untainted (low) perspective on a computation, tainted bits are ones whose values are unknown. Thus as a shorthand notation, we can represent tainted values in a modified binary representation with three kinds of digits: `0`, for a bit that is untainted and 0; `1`, for a bit that is untainted and 1; and `?`, for a tainted bit. Thus `1?0` represents a number whose second bit is tainted; in effect, the value from the high perspective might be either 4 (binary `100`) or 6 (binary `110`).

A. Taint Propagation Rules in Practice

We make three important observations about the data-centric noninterference model. First, the model is defined using information flows between bits. Thus it directly describes systems in which taint is labeled per bit. Not all implementations take this approach, but the model extends naturally to coarser-grained taint. For example, there is information flow from byte x to byte y as long as there is information flow from any bit of x to any bit of y. Results from a coarse-grained analysis are inherently limited in their precision, but for any granularity, we can try to achieve the most precise results expressible at that granularity. In this paper we are interested in exploring the maximum possible precision, so we focus on bit-level tainting.

Second, we observe that the precision of taint results also depends on the granularity of the *computation* analyzed. The reason is that the taint status of bits does not include information about how some bits might be correlated with others. For instance, suppose we take a single tainted bit `?` (representing either 0 or 1) and multiply it by an untainted 3 (binary `11`). The result must be either 0 (`00`) or 3 (`11`); thus both the low bits should be tainted, represented as `??`. If we know where the value came from, we know that the first and second bit positions must have the same value, but this information is missing in the tainted-bit representation, which could equally well describe a 1 (`01`) or 2 (`10`). This inherent imprecision of the representation in turn leads to imprecision in later results. For instance if we take the tainted bit value `??` and multiply it by 3 again (i.e., ? x 3 x 3), the result is `????`, since there is information flow to each of the four bits of the result. On the other hand, if instead of multiplying it by 3 twice as two operations, we had started with the tainted bit `?` and multiplied it by 9 in one operation, the result would be the more precise `?00?`.

This is a general phenomenon: expressing a larger computation in terms of smaller ones and applying sound taint analysis to each operation separately will always give sound final results. However, applying precise taint analysis to each operation separately will often not give as precise a result as analyzing the entire computation at once.

At the binary level, there are two common choices for taint analysis: we can either perform the analysis and update the taint labels after each instruction, or we can translate each instruction into a sequence of operations in a simpler intermediate representation (IR), and analyze the taint effects of each IR operation separately. Though this IR-level approach has other advantages, it can come at a cost to precision for the reason described in the previous paragraph. As an instruction-level example, consider an instruction (such as the `BIC` instruction on ARM) which computes the bitwise-AND of one register and the bitwise negation of another: $z = x \wedge \overline{y}$. If the two inputs are the same register, this has the effect of clearing the output register, so if this instance of this instruction is analyzed as a unit, the output should be completely untainted. On the other hand, an IR-level taint analysis that treated the AND and NOT as separate operations would be unable to tell that one operand of the AND was the negation of the other, so the result would still be tainted. Our formal verification can reveal these kinds of imprecision. VITA analyzes information flows at the x86 instruction level. SPITA takes a hybrid approach which primarily leverages an IR, but also applies some instruction-level rules to improve precision, as we describe in more detail in Section IV.

A final remark is that as specified so far, the model does not place any further restrictions on the choice of the input state s; the specific selection comes from the context in which we are verifying a taint analysis. To analyze the taint propagation in a particular situation, we can specify a concrete value for s. For instance, we can use a program state encountered during testing; this is what our VITA tool does. On the other hand, in constructing rules for taint propagation, we would like them to work correctly in all situations, so we look for taint rules that will soundly and precisely capture information flow for any choice of s. In short, s is a free variable when constructing rules and s is concretized when verifying rules.

B. Verifying Taint Propagation Rules

Taint propagation rules have usually been defined based on domain expertise and then reasoned about manually, or simply left unverified due to the difficulties of manual verification. For example, Memcheck has many special case rules, but according to its project suggestions webpage, formal verification of the rules is still needed [28]. The concepts for formal verification of tainting rules are introduced in this section.

The most obvious representation for bit-level taint, used for instance by Memcheck, is to maintain taint bits parallel to data bits with the same structure: for instance, the taint information for a 32-bit data word is represented by another

32-bit word, with the first bit of the taint word reflecting the taint status of the first bit of the data word, etc. We adopt the convention that a taint bit value of `1` indicates that the corresponding data bit is tainted, while `0` indicates untainted. Memcheck uses the opposite convention in its implementation (for what are referred to as validity or "V" bits), but because of the duality of Boolean algebra, the choice makes little difference. We will use the suffix $\_t$ for variables holding taint; for instance $S\_t = S$ is the set of all possible taint states. The taint propagation rule for a given operation is a function that takes as inputs the data state before the operation and the taint state before the operation, and yields the taint state after the operation: $\text{rule}_{op} : S \times S\_t \rightarrow S\_t$. Our definition of a sound and precise rule is that the taint bit for an output position b should be set if (soundness) and only if (precision) there is an input bit position a for which there is information flow from a to b and a is tainted.

An equivalent perspective on the soundness of a rule, analogous to noninterference, is that for each bit position b that is untainted after an operation, it should be the case that for any choice of values for the tainted input bits, the value of that untainted output bit is constant. If this condition fails, and there is an output bit that is affected by the tainted input but is not itself tainted, we say that the rule suffers from a false negative error. As a formula, let y_t be the output taint after applying the rule for the operation f to the input data state x and the input taint x_t. We make the following definition.

Definition 2: A rule $y\_t = \text{rule}_f(x, x\_t)$ applied to an operation $y = f(x)$ has a *false negative* error if:

$$\exists b, x_1, x_2 : (y\_t|_b = 0) \wedge ((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \wedge (f(x_1)|_b \neq f(x_2)|_b)$$

for some bit position b. Equivalently we can also compare all the untainted output positions at once:

$$\exists x_1, x_2 : ((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \wedge ((f(x_1) \wedge \overline{y\_t}) \neq (f(x_2) \wedge \overline{y\_t})) \quad (2)$$

Conversely, a rule has a false positive error if there is a bit position which is tainted, but does not in fact depend on the tainted input:

Definition 3: A rule $y\_t = \text{rule}_f(x, x\_t)$ applied to an operation $y = f(x)$ has a *false positive* error if:

$$\exists b : (y\_t|_b = 1) \wedge \forall x_1, x_2 : ((x_1 \wedge \overline{x\_t}) = (x_2 \wedge \overline{x\_t})) \Rightarrow (f(x_1)|_b = f(x_2)|_b) \quad (3)$$

Observe that the input state variables x and x_t are free in Equations 2 and 3. When checking the taint propagation in a trace, we instantiate them with values taken from an execution. When checking the correctness of a rule in the abstract, we quantify over all possible values for x and x_t: a rule is sound if there is no value of x and x_t for which Equation 2 holds, and precise if there is no value of x and x_t for which Equation 3 holds.

III. Constructing Tainting Rules

In the previous section we presented a formal model for taint analysis based on noninterference, and defined soundness and precision based on information flow. Since, in security applications, unsoundness can lead to missed attacks (a result we consider worse than false alarms), we choose to first construct a set of rules that are guaranteed to be sound, and then refine them to maximize precision.

Correspondingly, this section is separated into three parts. First, we discuss how to construct sound rules by identifying all bit-wise information flows in operations. Second, we verify that the rules are indeed sound, and discuss precise rules as well as how to verify them. Finally we compare the resulting rules with previously published taint trackers. We draw our examples from the x86 instruction set, but the techniques and most of the specific rules are applicable to other architectures, since the same basic operations (such as addition and bit shifts) are provided by all CPUs.

A. Constructing Sound Rules

Recall a rule is sound if every information flow from a tainted input bit to an output bit is marked by making the output bit tainted. Thus, to construct a sound rule, we first identify all possible information flows in an instruction and then summarize the flows with a rule. Since definition 1 is a satisfiability problem, we use satisfiability-modulo-theories (SMT) solvers to identify the information flows. There are two stages. First, the behavior of each instruction of interest is modeled using the bitvector operations of SMT solvers. To maintain compatibility with a wide range of solvers, the instructions are modeled in SMT-LIB Version 2 [29] ("SMT2" for short). Hence, the output of the first stage is a collection of SMT2 files describing the instructions' behaviors. The second stage uses SMT solvers to identify all information flows.

Stage 1: Behavioral Definitions: Since there are many x86 instructions, we first divided the instruction set into four categories: *data transfer*, *control transfer*, *arithmetic and logic* and *special*. Data and control transfer instructions have simple semantics with obvious bitwise information flow relationships and do not warrant further analysis. The arithmetic and logic category includes instructions that are likely to be supported in any general-purpose architecture. We focus on these instructions for wider applicability. The rest of the instructions fall into the *special* category. We included such special instruction, *cmpxchg* in our tests because it has an unusual information-flow pattern.

In total we analyzed over 150 different arithmetic and logic instructions. After some initial tests, we found that the precise mnemonics and operand choices (e.g., `add r/m8, r8` vs. `add r8, r/m8` vs. `add r16/32, r/m16/32`), did not affect the information flow patterns. Thus, we decided to simply focus on generic 32-bit register instruction formats (e.g., `add dst, src`). Our 26-instruction test set is outlined in Table I.

Since the correctness of the behavioral definitions is paramount, we relied on both BAP [30] and the developer's

```
(define-sort STATE () (_ BitVec 70))
(define-fun dst ((S STATE)) (_ BitVec 32)
  ((_ extract 69 38) S)
)
(define-fun f_add ((S STATE)) (_ BitVec 32)
  (bvadd (dst S) (src S))
)
(define-fun f_of ((S STATE)) (_ BitVec 1)
  ((_ extract 31 31)
    (bvand ( bvxor (dst S) (src S) #xFFFFFFFF )
           ( bvxor (dst S) (f_add S) )
) ) )

;;;Other function definitions

(declare-const DST (_ BitVec 32))
(declare-const SRC (_ BitVec 32))
(declare-const ZF (_ BitVec 1))
;;;Other declarations

(assert (exists ((i (_ BitVec 32)) (j (_ BitVec 32)))
  (not (= (dst (add (concat DST i ZF OF SF AF CF PF)))
          (dst (add (concat DST j ZF OF SF AF CF PF)))
) ) ) )
(check-sat)
```

Figure 1: SMT2 Definition and Test for `add dst, src`

manuals to help define the models. Given an instruction, we wrote assembly code to exercise different aspects of its behavior, linked them into an executable, and then lifted the executable into BAP's internal IR (BIL). We then extrapolated a single SMT2 behavioral representation from the instruction instances cross-checked against the processor documentation[1]. In essence, the correctness of our models depends on the correctness of BAP and our ability to identify any errors or bugs by hand.

As a quick example, a portion of the SMT2 definition for `add` is shown in Figure 1. The figure shows that `add` has an SMT2 equivalent operation `bvadd` meaning it is a basic operation. The flags calculation logic was extracted from BAP.

Stage 2: From Information Flow to Sound Rules: The goal of this stage is to take the SMT2 files from stage 1 and identify all possible information flows. For each file, we iterate through all possible pairs of input and output bits and query Z3 for the satisfiability of the condition in Definition 1. An example query can be found at the bottom of Figure 1. The query is used to determine whether there are two values i and j of `src` such that the values of `dst` are different after `add`. In other words, whether there is information flow from `src` to `dst`. In this case, if the solver returns "sat" then it means there is information flow from the input to the output. "unsat" means there is no information flow. The bitwise query is more involved, however, it follows the same pattern.

The resulting statistics for all the instructions are summarized in the first five columns of Table I. The instructions are presented in the first column; the input operands, both implicit and explicit, in the second; output operands, both implicit and explicit, in the third; the total number of input-bit to output-bit combinations in column four; and the time it took for Z3 to process the queries is shown in column five. We used a new instance of Z3 for each test case and thus the timing results include process creation overhead.

As expected, logical operations return results extremely quickly whereas signed multiply and divide takes the most time. Overall, it took less than 14 hours on an Intel Core-i7 860 to automatically identify all information flow relationships for 26 arithmetic and logical instructions.

The Rules: Once all of the possible information flows were revealed, we then summarized the flows into simple rule types. The sixth column of Table I indicates the general flow type for each instruction. To identify the flow types, we first graphed the information flow results to increase understandability. Specifically we generated directed graphs, one per input bit, where the nodes are bits of the state S and edges signify the potential for information flow from the source bit to the destination bit. As an example, we combined the 32 graphs from the 32 input bits of `dst` in `or` to produce Figure 2. The bit to bit *in-place* information flow relationship is evident.

1) Information Flow Types: There are four distinct information flow patterns between the source and destination operands. These four flow types serve as four different sound rules that we will refine later on. Note that we did not find any patterns of interest for the flags.

1) *In-place:* Information can only flow from bit i of the source to bit i of the destination. The Memcheck name is *UifU*.
2) *Up:* Information can only flow from bit i of the source to bits j of the destination where $j \geq i$. Figure 3 depicts this behavior, showing the combination of the information flow graphs for bits 7, 20, and 31. It is evident from the figure that information only flows from bit 7 of the source operand to bit 7 and higher of the destination. The same applies to bits 20 and 31, where bit 31 of the source only flows to bit 31 of the destination. The Memcheck name is *Left*.
3) *Down:* Dual to up, information can only flow from bit i of the source to bits j of the destination where $j \leq i$. Not used in Memcheck, but if added it should be called *Right*.
4) *All-around:* Information can flow from bit i of the source to any bit of the destination. The Memcheck name is *Lazy*.

We stress that there are times when a single instruction requires multiple tainting rules. Table I is not an exhaustive list. The divide instructions are good examples of this. In the divide operation, `edx:eax` is divided by `rm32`, the quotient placed into `eax` and remainder into `edx`. Intuitively, division is similar to shift right and thus the flow type for `edx:eax` to `eax` should be *down*. On the other hand, the flow type for `edx:eax` to `edx` is *all-around* since nothing definitive can be said about the relationship between the divisor and the remainder without concrete value analysis.

2) `bsf`*,* `bsr` *and* `cmpxchg`*:* `bsf` and `bsr` are two instructions that iterate through bit positions to find the first 1-bit and have internal control flow behaviors, leading to potentially erroneous manually defined policies. To illustrate

[1]Godefroid and Taly present algorithms to automatically generate similar behavioral specifications [31]

						Previously Published Taint Trackers								
Instruction	**Inputs**	**Outputs**	**# Cases**	**Runtime**	**Flow Type**	**DroidScope[16]**	**Cat1**	**libdft[9]**	**Minemu[10]**	**Cat2**	**TEMU[15]**	**Cat3**	**Memcheck[32]**	**SPITA**
adc *dst, src*	dst,src,cf	dsr,src,zf,of,sf,af,cf,pf	4550	1m19s	U	**A**	**A**	I	**A**	**A**	S	S	U	S
add *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m13s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	S	S
and *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	I	**A**	I	I	S	S	S
dec *dst*	dst	dst,zf,of,sf,af,pf	1184	20s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	U	S
div *rm32*	edx,eax,rm32	edx,eax,rm32	9216	95m48s	D	**A**	**A**	I	N	**A**	**A**	**A**	**A**	D
idiv *rm32*	edx,eax,rm32	edx,eax,rm32	9216	307m04	A	A	A	I	N	A	A	A	A	A
imul1 *rm32*	eax,rm32	edx,eax,rm32,of,cf	6272	289m51s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
imul2 *dst, rm32*	dst,rm32	dst,rm32,of,cf	4224	52m37s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
imul3 *dst, rm32, imm32*	rm32,imm32	dst,rm32,imm32,of,cf	6272	53m56s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
inc *dst*	dst	dst,zf,of,sf,af,pf	1184	19s	U	**A**	**A**	I	**A**	**A**	**A**	**A**	U	S
mul *rm32*	eax,rm32	edx,eax,rm32,of,cf	6272	16m02s	U	**A**	**A**	I	N	**A**	**A**	**A**	U	U
not *dst*	dst	dst	1024	15s	I	**A**	I	I	**A**	I	I	I	I	I
or *dst, src*	dst,src	dst,src,zf,sf,pf	4288	1m05s	I	**A**	I	I	**A**	I	I	S	S	S
rcl *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	A	N	A	A	A	A	A	S
rcr *dst, imm8*	dst,imm8,cf	dst,imm8,of,cf	1722	42s	A	A	A	N	A	A	A	A	A	S
rol *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	A	N	A	A	A	A	S	S
ror *dst, imm8*	dst,imm8	dst,imm8,of,cf	1680	41s	A	A	A	N	A	A	A	A	S	S
sal *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	U	**A**	**A**	N	**A**	**A**	S	S	S	S
sar *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	34s	D	**A**	**A**	N	**A**	**A**	S	S	S	S
sbb *dst, src*	dst,src,cf	dst,src,zf,of,sf,af,cf,pf	4550	1m21s	U	**A**	**A**	I*	**A***	**A**	**A**	**A***	**A**	S
shr *dst, imm8*	dst,imm8	dst,imm8,zf,of,sf,af,cf,pf	1840	35s	D	**A**	**A**	N	**A**	**A**	S	S	S	S
sub *dst, src*	dst,src	dst,src,zf,of,sf,af,cf,pf	4480	1m17s	U	**A**	**A**	I*	**A***	**A***	**A***	**A***	S	S
xor *dst, src*	dst,src	dsr,src,zf,sf,pf	4288	1m05s	I	**A**	I	I*	**A***	**A***	**A***	**A***	I	I
bsf *dst, src*	src	dst,src,zf	2080	31s	A	N	A	I	N	A	A	A	A	S
bsr *dst, src*	src	dst,src,zf	2080	31s	S	N	**A**	I	N	**A**	**A**	**A**	**A**	S
cmpxchg *rm32, r32*	eax,rm32,r32	eax,rm32,r32,zf,of,sf,af,cf,pf	9792	2m39s	S	N	**E**	**E**	N	**E**	**E**	**E**	**E**	S
TOTAL			102064	13h52m48s										

Table I: Flow Type Results for x86 Instructions
Flow Types: (U)p, (D)own, (I)n-place, (A)ll-around, (S)pecial, (N)ot-Supported, (S)pecial, (E)ax is tainted in `cmpxchg`,
* - Zeroing Idiom, **Boldface** - Generated Policy is more precise

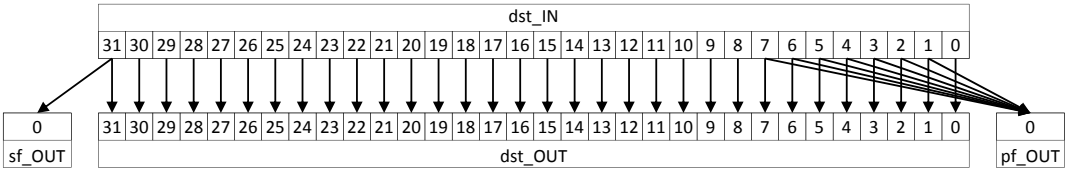

Figure 2: Information flows of *dst* in the `or` instruction

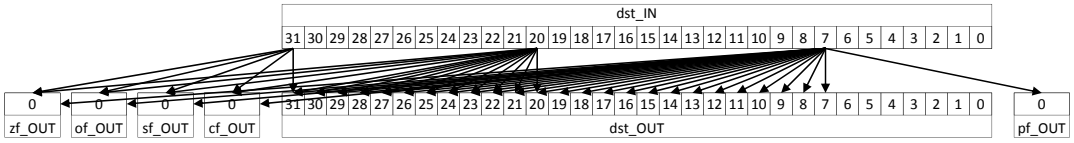

Figure 3: Information flow of bits 7, 20 and 31 of `dst` in `sbb`

the value of our approach, the information flows for input bits 1, 5 and 8 are depicted in Figure 4. Since `bsf` tests the lowest bits first, all subsequent iterations of the loop depend on the lower bits and thus the flow behavior is equivalent to propagating all-around. For example, if only bit number 5 of `src_IN` is 1, then bits 0,1 and 2 of `dst_OUT` are set. However, if bit number 1 is also set, then only bit 0 of `dst_OUT` is set. Thus, the setting of bits 1 and 2 depends on the fact that bit 1 of `src_IN` is 0. `bsr` scans from the highest bit position down, thus the flow pattern is much more direct and is special, not all-around. It is noteworthy to point out that some of these complex behaviors can also be identified as control flow dependencies using the technique

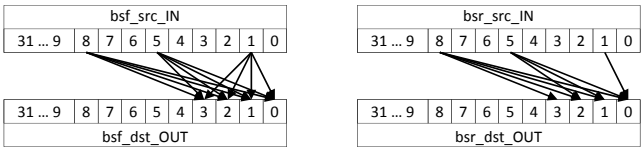

Figure 4: Comparison between *bsf* and *bsr*

proposed by Ferrante et al. [33] (both Dytan and DTA++ use this technique). However control flow dependencies are traditionally analyzed per variable and therefore would not reveal the bit-level relationships of `bsr` as shown in Figure 4.

```
1. cmpxchg (rm32, r32) {
2.   if (eax == rm32 ) then
3.     rm32 = r32;
4.   else eax = rm32;
5. }
```

Figure 5: Pseudocode for `cmpxchg` (flags are omitted)

`cmpxchg` (Figure 5) is a special x86 instruction that demonstrates how IR tainting (i.e., emulating the cmpxchg instruction using IR instructions and then tracking taint through the IR instead of cmpxchg as a whole) can lead to false-positives. It also shows the benefit of automated analysis. Applying Definition 1 to the instruction as a whole shows that there is no information flow from `eax` to `eax` because the output value of `eax` is fully dependent on the input value of `rm32`. On the other hand, if information flow was analyzed line-by-line, and thus mimicking the behavior of IR based tainting where each line is a corresponding IR, `eax` will be tainted if `eax` was tainted before the instruction. This is because `eax` was unchanged in the equals branch (line 3) and thus retains its taint. The case for simple control flow dependencies is even worse. Since `eax` is used in the comparison on line 2 and also as an l-val on line 4, it will remain tainted in the not-equals branch. All in all, care must be taken to ensure that IR level tainting does not introduce false positives or false negatives and that the implementation is formally verified to be correct.

B. Constructing Precise Rules

The previous section focused on the construction of sound rules. We arrived at four basic rules that are sound by construction. However, we ended the section with a discussion of `cmpxchg`, which motivated the need for formal verification of tainting rules. Tainting rule verification is accomplished in two steps: the operation and the tainting rules under test are formally specified, and then solvers are used to determine whether Equations 2 and 3 are satisfiable. The formal specification step is straightforward using the models from Stage 1. Only the rules to be verified remain to be modeled. We continue to use the SMT2 language and the Z3 solver.

When we verified the sound rules from the previous section, we found that while all of the rules were sound (as expected), many of them were not precise. In order to construct precise rules, we sought inspiration from Memcheck, since it has many specially-defined rules. Our first step was to formally verify the soundness and precision of Memcheck's specially-defined rules. Memcheck's rules are defined for its own VEX IR and not directly for x86 instructions. However, both are bitvector operations and thus one can be translated into the other and vice versa. The Memcheck rules were obtained both from the original paper [32] as well as the Memcheck source code.

We were not always happy with Z3's performance, so we also supplemented it with a complementary decision procedure MONA [25]. MONA supports a different logic and uses a different solving approach, translating a restricted subset of second-order logic into finite state machines whose transition tables are in turn encoded as BDDs. MONA has the advantage that it can naturally express properties over unbounded bitvectors, so we need not fix a word size in advance. MONA also deals more gracefully with alternating quantifiers. By contrast in Z3 there was large performance gap between soundness queries, which are pure satisfiability, and precision queries which require a $\forall$ quantifier inside a $\exists$. On the other hand MONA is less expressive, and is not compatible with the SMT2 format; we had to manually construct models and rules. We represent bitvectors to MONA as a sets of non-negative integers, where an integer is a member of the set if the corresponding bit position is 1. Some operations have a natural counterpart in this representation (bitwise AND is set intersection) but some (such as addition) require complex definitions, and because MONA allows only finite sets, negation and bitwise NOT cannot be directly represented at all.

The verification results are summarized in the top portion of Table II. There are five columns: the operation, Z3 result for soundness, the Z3 result for precision and finally, if the Z3 result was inconclusive (i.e., Z3 did not return a result after 24 hours of processing), the MONA result of whether the rule is precise, and the corresponding rule that we verified.

As the results show, all of the special rules defined in Memcheck are sound for operands up to 256 bits[2] Additionally, the special rules for `and` and `cmpEq` are also precise up to 256 bits. The same cannot be said for the other rules though. In particular, the special rule for `or`, which is similar to the one for `and`, has only been verified for 2-bit operands. Operands of greater length, e.g., 4 bits, are inconclusive since Z3 timed out. (This performance is surprising since `or` is one of the simpler rules, and analogous to the `and` rule which performed well, but we have not yet isolated the cause.)

The rest of the Z3 results show that for most cases, Z3 times out for operands beyond 16 bits in length. Since the tests for smaller bit lengths returned quickly, we hypothesize that the size of the state space to explore is the culprit. In these cases we proceeded to use MONA to determine precision. The results show that MONA was able to verify precision of the `add`, `or` and `sub` rules.

All of the shift rules were shown to be imprecise. This is because the shift amount can be tainted. In this case, the PCastYX function simply marks all bits of the output as tainted. Subsequently, we asserted that the shift amount is not tainted, and re-verified the rules. They were shown to be precise for up to 16 bit operands using Z3.

New Rules: Since the Memcheck rules were verified to be both sound and mostly precise, we incorporated them into SPITA. We also sought to define new rules to handle additional cases. The new rules are summarized in the bottom part of Table II. The rules for `adc` and `sbb` are essentially three-operand versions of the precise rules for

[2]We chose 256 bits as the maximum length to test, since we are unaware of any architectures with operands greater than 256 bits.

Operation	Sound	Precise Z3	Precise MONA	Rule (in SMT2)
add	256	16	✓	(bvor (bvor v1 v2) (bvxor (bvadd d1_min d2_min) (bvadd d1_max d2_max)))
and	256	256	✓	(bvand (bvor v1 v2) (bvand (bvor d1 v1) (bvor d2 v2)))
cmpEq	256	256		(ite (= (bvor (bvor v1 v2) (bvnot (bvxor d1 d2))) #xFFFFFFFF) (ite (= (bvor v1 v2) #x00000000) #b0 #b1) #b0))
or	256	2	✓	(bvand (bvor v1 v2) (bvand (bvor (bvnot d1) v1) (bvor (bvnot d2) v2)))
rol	256	16*		((bvor (PCastYX v2) (rol v1 d2))
ror	256	16*		((bvor (PCastYX v2) (ror v1 d2))
sal/shl	256	16*		((bvor (PCastYX v2) (bvshl v1 d2))
sar	256	16*		((bvor (PCastYX v2) (bvashr v1 d2))
shr	256	16*		((bvor (PCastYX v2) (bvlshr v1 d2))
sub	256	4	✓	(bvor (bvor v1 v2) (bvxor (bvsub d1_min d2_max) (bvsub d1_max d2_min))))
adc	256	16	✓	(bvor (bvor v1 v2) (bvxor (bvadd d1_min d2_min cf_min) (bvadd d1_max d2_max cf_max)))
rcl	256	16*		(bvor (PCastYX v2) (rcl v1 d2 vcf)) where vcf is the taint value for the carry flag
rcr	256	16*		(bvor (PCastYX v2) (rcr v1 d2 vcf)) where vcf is the taint value for the carry flag
sbb	256	4	✓	(bvor (bvor v1 v2) (bvxor (bvsub d1_min (bvadd d2_max cf_max)) (bvsub d1_max (bvadd d2_min cf_min))))
bsf	32		16^z	Destination is tainted if (bvule (bsf v1) (bsf d1))
bsr	32		16^z	Destination is tainted if (bvuge (bsr v1) (bsr d1))

Table II: Precise Rules and Verification Results: Length of operands verified (in bits).
✓Verified for all lengths. * Shift amount is untainted. ^{z} Non-zero operand for `bsf, bsr`; precise rule omitted for space. d are the operands and v are the shadow taints. The *min* terms represent the value where all tainted bits are set to 0, e.g. d1_min = (bvand d1 (bvnot v1)). The *max* terms represent the value where all tainted bits are set to 1, e.g. d1_max = (bvor d1 v1). The *PCastYX*, pessimistic cast, function returns 0 if the operand is 0 and 1s in all bit positions otherwise.

`add` and `sub`. Similarly, the rules for `rcl` and `rcr` are natural extensions of the shift and rotate rules identified in Memcheck. Intuitively, this makes sense since these two operations are simply the $n + 1$ versions of the `rol` and `ror` operations respectively. The intuition behind the `bsf` and `bsr` rules is that if an untainted bit is already 1, then it does not matter what the value of the tainted bit is as long as the tainted bit is scanned after the untainted bit.

It is important to note that these instructions do not have equivalent operations in Memcheck. To put it differently, Memcheck will use VEX IR operations to emulate this functionality, and thus these rules cannot be directly incorporated into Memcheck. The same applies to SPITA since taint is propagated at the TCG IR level. To ensure that taint is propagated precisely, these rules are implemented at the target instruction level (e.g., x86). That is, when SPITA sees one of these instructions, it will use the special rules to propagate taint and disables TCG IR tainting for the IR that is used to emulate the instruction.

C. Comparing With Previous Policies

Two sets of rules were generated in the previous sections: the automatically constructed sound rules, and the verified mostly precise rules—summarized in the "Flow Type" and "SPITA" columns of Table I respectively. We now compare them to the rules used in 19 taint analysis systems found in the literature. They all track taint through bitvector instructions. To condense the results, we separated the implementations based on the sophistication of their rules. There are three categories. Cat1 includes all implementations that rely on simple l-val r-val relationships [34], [7], [10], [16], [17], [20], [23]. Effectively, their rules consist of *in-place* and *all-around* flow types only.

Cat2 includes systems with simple sanitization rules such as zeroing idioms (e.g., `xor eax, eax`) [9], [35], [8], [14], [18], [19] and Cat3 includes those with more sophisticated propagation rules [1], [15], [2], [21], [22].

The results are shown in the middle columns of Table I. Since tainting rules can be updated after publication, all of the implementations that we were able to find source for have their own column showing the policy defined in the source. The policies are ordered according to the expected number of false positives with the least precise on the left (column 7) and the most on the right (column 14).

As the results show, many of the rules used in previous publications are less precise than the automatically generated sound policies. In fact, none of the policies used the propagate-up flow type of `sbb`. Another interesting point is that the default rules for libdft and Minemu are not sound.

Either some of the instructions were not supported, or in the case of libdft, the default use of union or *OR* leads to false negatives in instructions such as `add` where taint should propagate up. Finally, the results indicate SPITA has the most special, and proven precise, rules of any implementation.

IV. Implementing SPITA

The outcome of the previous section was a collection of taint propagation rules that are sound. Most of them are precise as well. These results were implemented in a new taint tracker named SPITA (Sound and Precise Instruction-level Taint Analyzer). SPITA is designed as a hybrid instruction level taint analysis platform since it tracks taint at the bit-level through QEMU's TCG IR operations for simple instructions (e.g., and, or, add, sub, etc.), but tracks taint taint through target instructions (e.g., x86 instructions) for complex ones (e.g., bsf).

In order to support taint analysis, we first created a shadow system state (i.e., shadow memory and registers) to store the taint labels. Since we label taint at the bit level, the shadow memory and registers essentially double the resource requirements of the QEMU guest. This also means that the shadow memory is organized into page-table structures just like real memory.

To propagate taint, the basic idea is to ensure that for each TCG IR instruction that is used to emulate a target instruction, additional TCG IR instructions will be added to propagate the taint. In other words, for each instruction that operates on real operands, we insert a sequence of instructions that operate on the corresponding taint operands.

Figure 6 illustrates how we implement precise tainting for an x86 instruction, `add eax, ebx`, at TCG IR level. QEMU will translate this x86 instruction into 4 TCG IRs (shown in Figure 6(a)): two CPU registers are first moved into two temporary variables; a sum is calculated on the temporary variables; and finally the sum is moved into `eax`. For each of these original IR instructions, we insert one or more IRs to implement the tainting logic, which is shown in Figure 6(b). We will create shadow variables for general-purpose CPU registers, such as `taint_eax` for `eax` and `taint_ebx` for `ebx`. For temporary variables, we will also create new temporary variables to shadow them. A `mov` operation can be simply shadowed by another `mov`. For example, we insert `mov_i32 tmp42, taint_eax` to shadow `mov_i32 tmp5, eax`. To apply the precise tainting rule for `add` on L16, we need to insert 11 IRs (from L5 to L15).

Note that in most cases we insert the taint IRs before the original IR, because most precise tainting rules need to check the operands' concrete values. The concrete values may have changed after the execution of the original IR. Of course, we need to take care of IRs that may raise exceptions and fix up the labels (if any) so that they point to the beginning of tainted IRs instead.

Using taint rules defined for TCG IR is generally more convenient because there are fewer IR instructions than

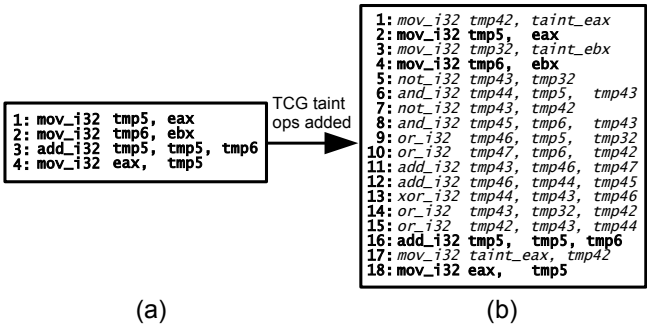

Figure 6: An example of implementing precise tainting for `add eax, ebx` at TCG IR level.

x86 instructions. A smaller number of core rules can be automatically applied to cover a much larger number of x86 instructions. An IR-level implementation also facilitates future ports to additional architectures. However purely-IR-level rules can be less precise, for the reasons discussed in Section II-A. For this reason, SPITA also includes support for triggering more specific taint rules based on a pattern of IR statements. For instructions that use helper functions, we insert the tainting rule directly into the functions. For instance, `bsf` and `bsr` are implemented as helper functions defined in C. We simply implemented the tainting rule in C as well and inserted the new code block into the helper function.

After we implemented tainting rules at the IR level and carefully scrutinized the helper functions, SPITA now has the full tainting support for all integer-based x86 instructions and a few floating point and SSE instructions with simple semantics (totaling 369 opcodes - operand types and widths ignored). A complete list of mnemonics with respect the soundness and precision guarantees can be found in Table III.

Table III: **SPITA supported x86 instructions.** The tainting rules for all these instructions are sound, and most are also precise. The imprecise ones are marked with "*".

AAA* AAD* AAM* AAS* ADC ADD AND ARPL BOUND BSF BSR BSWAP BTC BTR BTS BT CALLF CALL CBW CDQ CLC CLD CLI CLTS CMC CMOVB/CMOVC/CMOVNAE CMOVBE/CMOVNA CMOVL/CMOVNGE CMOVLE/CMOVNG CMOVNB/CMOVAE/CMOVNC CMOVNBE/CMOVA CMOVNL/CMOVGE CMOVNLE/CMOVG CMOVNO CMOVNP/CMOVPO CMOVNS CMOVNZ/CMOVNE CMOVO CMOVP/CMOVPE CMOVS CMOVZ/CMOVE CMPS CMPXCHG8B CMPXCHG CMP CPUID CWDE CWD DAA* DAS* DEC DIV ENTER FWAIT/WAIT HINT_NOP HLT IDIV* IMUL* INC INS INT0 INT1/ICEBP INT INVD INVLPG IN IRET JB/JNAE/JC JBE/JNA JCXZ/JECXZ JL/JNGE JLE/JNG JMPF JMP JNB/JAE/JNC JNBE/JA JNL/JGE JNLE/JG JNO JNP/JPO JNS JNZ/JNE JO JP/JPE JS JZ/JE LAHF LAR LDS LEA* LEAVE LES LFS LGDT LGS LIDT LLDT LMSW LOCK LODS LOOPNZ/LOOPNE LOOPZ/LOOPE LOOP LSL LSS LTR MOVBE MOVSX MOVS MOVZX MOV MUL* NEG NOP NOT OR OUTS OUT POPAD POPA POPFD POPF POP PUSHAD PUSHA PUSHFD PUSHF PUSH RCL RCR RDMSR RDPMC RDTSCP RDTSC REPNZ/REPNE REPZ/REPE REP RETF RETN ROL ROR RSM SAHF SAL/SHL SAR SBB SCAS SETB/SETNAE/SETC SETBE/SETNA SETL/SETNGE SETLE/SETNG SETNB/SETAE/SETNC SETNBE/SETA SETNL/SETGE SETNLE/SETG SETNO SETNP/SETPO SETNS SETNZ/SETNE SETO SETP/SETPE SETS SETZ/SETE SGDT SHL/SAL SHLD SHRD SHR SIDT SLDT SMSW STC STD STI STOS STR SUB SUB SYSENTER SYSEXIT TEST VERR VERW WBINVD WRMSR XADD XCHG XLAT/XLATB XOR

In this implementation, SPITA does not create shadow memory for the FPU stack and the MMX stack, and we do not have special tainting rules for instructions that operate on these stacks. This is a design decision common in security applications. We leave it as a future work to investigate sound and precise tainting rules for the floating point and SSE instructions. SPITA has been open-sourced. Readers can refer to the source code in the project homepage (ANONYMIZED FOR SUBMISSION) for the details of the exact tainting rules and the instruction coverage.

A. Evaluation

We evaluated SPITA on two fronts. The first test determines how much performance overhead can be expected from our additional TCG IR instructions and the second test determines what the impact on the translation block size is. All of these tests were executed within a Windows 7 virtual machine image running under SPITA. SPITA was executed on a 64-bit Ubuntu 12.04 Linux system (3.2.0 kernel) with a 32-core 2.0GHz Intel Xeon ES-2650 CPU and 128GB of RAM. 4GB of RAM was allocated to the VM environment.

The CPU benchmark results were gathered using the SPEC CINT2006 benchmark suite [36] and are summarized in Figure 7. Test 462.libquantum was omitted from the benchmark due to its incompatibility with Visual Studio 2010. The figure depicts the performance overhead of SPITA (with the additional taint propagating IR instructions) as compared to the baseline QEMU without any changes. It can be seen that the overhead for CPU-bound activities ranged from a high of 775% (464.h264ref) to 285% (429.mcf). Both the bitfield and FP emulation tests rely heavily on arithmetic operations, which as can be seen by the addition example above, require at least four additional IR instructions to propagate the taint. Thus, these results are not unexpected. As a juxtaposition of these results, we can also see that benchmarks that rely primarily on memory accesses, which require only one additional instruction to load the shadow taint memory, experience very low overhead (429.mcf, 471.omnetpp). This is also expected since loading the tainted memory only requires a single additional instruction. Furthermore, since paging is utilized and taints are expected to be sparse, the page table lookup and TLB lookups for the shadow memory is expected to be faster than regular guest memory.

We used the internal QEMU profiler ("info jit") to obtain the translation block (TB) statistics and determine the overhead due to the addition of the taint propagation IR instructions. For this evaluation, we executed a series of basic system tasks such as booting the Windows 7 guest, performing Google searches using the Chrome web browser, and manipulating text files. For the QEMU baseline, we found that the average TB contains 45.3 IR instructions with the largest TB having 464 instructions. An average of 29.3 temporary registers was used by the TBs, with a maximum 68 temporary registers used. On the other hand, SPITA TBs have an average of 86.7 IR instructions with the largest TB containing 520 instructions. On average, 74.0 registers were used with a maximum of 358 temporary registers.

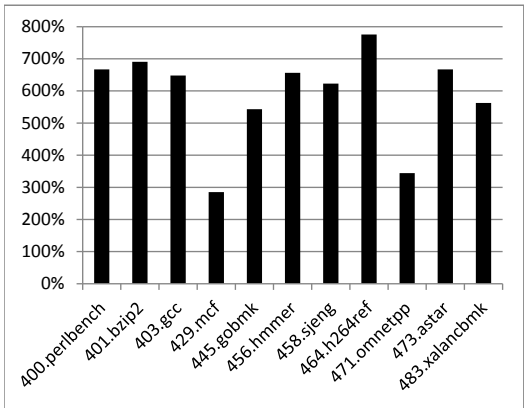

Figure 7: Overhead for SPEC CINT2006 benchmark suite

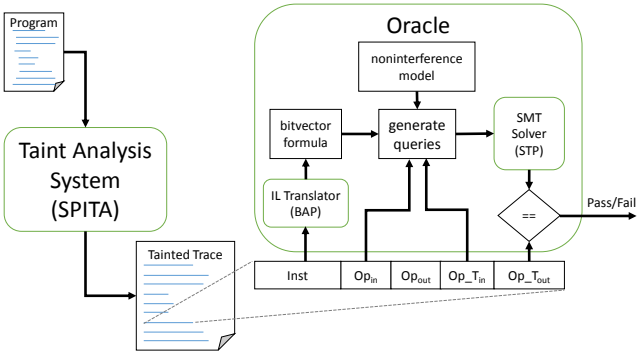

Figure 8: Per-Trace Verification Overview

In short, these results show a reasonable, but non-negligible increase in TB size. This increase can also adversely impact performance since having larger TBs means less TBs can be available in memory at a time, which in turn leads to more translation cache flushes, reducing performance. Additional tests, designs and improvements (such as increasing the cache size) is left as future work.

V. Per-Trace Verification

The previous section discussed SPITA, a new instruction-level taint tracker that is based on QEMU. In this section, we present *per-trace verification* as a technique to verify the correctness of a taint analysis system's implementation. A high level overview of the process is depicted in Figure 8.

In per-trace verification, the taint analysis system under test (e.g., SPITA) executes a program and generates a tainted trace. The trace is a log of all the instructions executed along with additional metadata. Each entry contains the instruction executed, the input operand values, the output operand values, and the corresponding taint label assignments. (A sample log entry can be found in Figure 10)

For each entry in the instruction trace, an *oracle* is used to determine whether the resulting taint matches the noninterference model. The oracle consists of three main components. An *IL translator* is used to translate the operation (add in the example) into a bitvector formula. A *query generator* then takes the translated formula, the concrete

values from the trace entry, and the input taint assignments, and generates a query to determine the correct output taint labels. This query is subsequently sent to an SMT solver and the results compared to the output taint as recorded in the trace entry. If they agree then implementation is correct for this particular operation and machine state. If they disagree either the rule is imprecise or there is an implementation bug.

Per-trace verification has a number of advantages. First, the traces can be generated and verified independently and thus processed in parallel. Second, the problem of verifying traces one instruction instance at a time is more tractable: using concrete values reduces the state space to explore. Third, the oracle can also be used as a taint analysis system itself. For example, a taint analysis system might use sound but imprecise tainting rules to improve runtime performance and then use the oracle to reprocess the trace offline and remove any false positives.

The major limitation of per-trace verification is coverage. Per-trace verification will not be complete unless the traces used to verify the system cover all possible system states (i.e., all possible combinations of operations, operand values and taint values). To maximize coverage, we use a collection of over 600,000 test programs from the PokeEMU [26] project. These test programs were automatically generated by exploring all of the different instruction decode and execution paths of the Bochs x86 emulator. They provide full path coverage of more than 800 protected-mode x86 instructions, and so our per-trace verification results inherit this same extensive coverage.

Implementation: In order to verify SPITA, we first implemented an instruction tracer to generate the tainted trace. We implemented the oracle using BAP as the IL translator and STP [37] as the SMT solver (Z3 works as well). Specifically we express the bitvector formula and queries in the BAP IR, allowing us to use BAP's existing interface to STP (or Z3).

The query generator is best explained with an example. Figure 9 shows a simplified version of the query generated for the `xor ebx, ebx` x86 instruction. Line 7 shows the BAP IR that assigns ebx to 0. This is the bitvector formula for the operation that was generated by BAP. Lines 2 and 3 show the input value of ebx as well as its taint labels. The final goal condition to query—are there two assignments of tainted bits such that bit 31 of the output is not equal?—is defined in line 13. The two compared variables `goal1` and `goal2` each represent the state of bit 31 of ebx after the operation, but they differ in the position of a free variable (`R_EBX_01` or `R_EBX_02` respectively) which represents the varying tainted bit assignments.

The same basic logic of establishing two separate goals with two different free tainted bit assignments using the same emulation logic is used for all of the instructions recorded in the trace. This setup works well except for two cases. First, there are some behaviors that are undefined. For example, `bsf` has undefined behavior when the operand to scan is 0. There is no natural way to represent undefined behavior in our model; our choice so far has been to represent undefined behavior with no-op behavior. For `bsf` on 0, the destination operand remains unchanged. The second case is tainted pointer operands. When a memory operand has an untainted address, the single location referred to can be translated similarly to a register. But a tainted address could refer to many locations or even all of memory, which cannot be practically represented in a single BAP formula. This case is related to the more general difficulties of taint analysis and pointers described by Slowinska and Bos [38]. However we consider this issue orthogonal to our main correctness concerns since tainting rules are generally defined based on instructions and their operands (e.g., destination and source) and not on their operand types (e.g., register and memory).

```
 1. // Query for bit [31] of R_EBX:u32
 2. R_EBX_C:u32 = 0x46018902:u32
 3. R_EBX_T:u32 = 0x56718e20:u32
 4. //Concretization of flags
 5. goal:bool = false
 6. R_EBX:u32 = (R_EBX_O1:u32 & R_EBX_T:u32)
              | (R_EBX_C:u32 & ~R_EBX_T:u32)
 7. R_EBX:u32 = 0:u32 // sets R_EBX to 0
 8. //BAP IR for calculating the flags for xor ebx, ebx
 9. goal1:u32 = R_EBX:u32 & 0x80000000:u32
10. R_EBX:u32 = (R_EBX_O2:u32 & R_EBX_T:u32)
              | (R_EBX_C:u32 & ~R_EBX_T:u32)
11. //Same BAP IR for emulating xor
12. goal2:u32 = R_EBX:u32 & 0x80000000:u32
13. goal:bool = goal1:u32 <> goal2:u32
```

Figure 9: Query to determine whether bit 31 of EBX should be tainted

Verification of SPITA: The correctness of SPITA's rule implementations was verified using the over 600,000 PokeEMU test cases. Each test case was executed using SPITA and all instructions executed were logged into a tainted trace, one per test case. Due to the sheer number of test cases, we did not exhaustively try all possible taint assignments to the program state. Instead, we assigned random taint values to the program state at the beginning of execution and allowed it to be propagated through the program.

Each trace was then passed through the VITA oracle to determine whether there are any differences in the output taints. If the verification fails, we manually reviewed the offending instruction in an attempt to track down the source of the failure. If a bug was found, we patched it and then re-ran the offending test case to ensure that the bug was patched. We also re-ran similar test cases to ensure that a new bug was not introduced. In total, it took over 16 days to complete the verification task by running 80 verification instances in parallel. Each trace took approximately 3 minutes to complete. This does not include the extra time needed to address a couple of bugs that we found.

There were two incorrectly implemented rules in SPITA (`and` and `add`). As it turns out, both errors are due to the same implementation mistake. A text version of the offending trace entry can be found in Figure 10. The figure shows the concrete values of the operands as well as the input and output taints. According to SPITA, the output taint was 0xe44ae761, which failed verification since the expected taint from VITA was 0xe64ae761. Notice that bit 25 is 0 but

```
Inst: and %eax, %ebx // ebx = eax & ebx
Inputs: eax = 0x84be2329, ebx = 0xaed66ce1
Outputs: ebx = 0x84962021
Input Taints: eax_t = 0x7369C667, ebx_t = 0xec4aff51
Output Taints: ebx_t = 0xe44ae761 //should be 0xe64ae761
```

Figure 10: Trace entry for `and` bug

should be 1.

As it turns out, this error was due to the way we inserted the extra TCG IR to propagate taint in SPITA. In the code for adding the propagation code for `and`, we incorrectly placed the propagation code after the original `and` operation. As a result, instead of using the concrete value of 0xaed66CE1 for ebx to calculate the taint, we used the result of ebx (0x84962021). In fact, this bug was pervasive in our implementation, and we didn't understand it until found out that ADD has the same problem. In general, this bug only surfaces if the destination operand is also a source operand, and the value written to the destination happens to affect the final taint calculation, meaning it depends on both the concrete values as well as the taint assignments. This is why in Figure 6 we put the taint IRs before each original IR to propagate taint properly.

VI. Experiments with Realword Workloads

In this section, we evaluate how SPITA performs under realworld workloads. In particular, we like to compare SPITA with a previously published taint analysis tool under the same workloads. We choose to compare with TEMU [39] for two reasons. First, TEMU is a popular open-source dynamic binary analysis platform in the BitBlaze project [40], and many security projects have been built on top of it. Second, according to our verification in Section III-C, the tainting rules in TEMU are mostly sound and some special rules are applied to improve precision. By comparing SPITA with TEMU under real workloads, we can empirically demonstrate how much benefit SPITA can offer by being significantly more precise.

We ran a set of workloads in Windows XP Service Pack 3 and Linux 2.6.20 respectively. These workloads are tainted shell commands: we sent tainted keystrokes as commands to a shell and observed how each of the tainted commands was processed in the operating system. As the keystrokes must go through a series of code conversion to become valid characters, we enable the pointer tainting in system wide. It means that when a memory index for a memory read is tainted, we will mark the value to be tainted.

Table IV lists the results for this set of workloads. The top half of the table shows the shell commands for Windows, and the bottom half shows the ones for Linux. For each command, after it finishes execution, we observe the number of tainted byte in the main memory and the occurrence when `EIP` becomes tainted. We can see that for all the commands in Windows, the number of tainted bytes in SPITA is much smaller than the number in TEMU, demonstrating the benefit of SPITA being more precise. No tainted EIP was observed in either system.

Table IV: **Comparing SPITA with TEMU on tainted shell commands.** "n / m" indicates that "n" bytes are tainted, and "m" tainted EIPs are observed.

Command	SPITA	TEMU
`dir`	207 / 0	639 / 0
`cd`	146 / 0	616 / 0
`cipher c:`	929 / 0	3617 / 0
`echo hello`	660 / 0	3808 / 0
`find "jone" a.txt`	967 / 0	5684 / 0
`findstr /s /i jone ./*`	945 / 0	1333 / 0
`ls`	350 / 3	34923 / 0
`cd`	306 / 3	301 / 0
`cat ./readme`	545 / 31	26619 / 0
`echo hello`	744 / 9	704 / 0
`ln -s a.txt nbench`	1122 / 35	24707 / 0
`mkdir test`	551 / 9	23766 / 0

The results for the commands in Linux are somewhat different. Although the number of the tainted bytes marked by SPITA was generally much smaller than the one by TEMU, SPITA reported tainted EIPs for all these commands, whereas TEMU reported none.

These results look contradictory to the claim that SPITA should be more precise. So we manually examined and compared the taint propagation logs for the execution of the `cd` command generated by SPITA and TEMU. We did not finish examining all these tainted EIPs (totally 90), but we confirmed that at least a large portion of them were indeed correct. A common case is that a tainted character (the one we entered) was used as an index to call a function in a function pointer table. We found the same instruction sequences in the trace generated by TEMU. It means that TEMU in fact has under-tainting problem in its implementation, even though the tainting rules are generally sound by design.

Furthermore, to test the effectiveness of SPITA, we collected a set of malware samples that are known to have key-logging functionality. These sample set has 117 malware samples in total, spanning 29 malware families. The detail is listed in Table V. By sending tainted keystrokes into the guest system and observing if any untrusted code module accesses the tainted data, we can detect keylogging behavior. SPITA successfully detected the keylogging behaviors in all these samples, demonstrating the correctness of taint analysis in SPITA.

VII. Discussion

Dependencies: Our approaches to generating and verify policies are based on decision procedures, and our performance is wholly dependent on them. As in the experiments sometimes even simple-seeming queries can show disappointing performance. However, since SMT solvers use standard interfaces, they can improve independently and we can substitute different tools with no changes to the techniques in this paper. Our current results reflect fully automatic decision procedures, but it is also possible to get improved performance in exchange for more manual effort: for instance by adding witness (Skolem) functions to replace quantifier alternation.

Table V: **Keylogger Samples Set**

Keylogger Name	No
Backdoor.Win32	8
Email-Flooder.Win32.Webhat	1
Email-Worm.Win32	5
Gen:Application.Keylog	2
HEUR:Trojan.Win32.Generic	13
MonitoringTool:Win64/KGBKeylogger	2
Monitor.Win32.HomeKeylogger	2
Monitor.Win32.Perflogger	4
Monitor.Win32.Ardamax	1
Rootkit.Win32.Agent	1
Trojan.Win32.Agent	4
Trojan-Spy.Win32.Keylogger	39
Trojan.Win32.Scar.cone	1
Trojan-PSW.Win32.VB	2
Trojan-PSW.Win32.LdPinch	1
Trojan-Downloader.Win32	6
TrojanDropper:Win32	2
Trojan.Win32.Genome	6
Trojan.Win32.VB	2
Trojan-GameThief.Win32.Nilage	2
Trojan-Spy.Win32.Delf	2
Trojan-Spy.Win32.Small	1
Trojan-FakeAV.Win32.Agent	1
Trojan-Spy.MSIL.Agent	3
Trojan.Win32.Orsam	1
Worm.Win32.AutoRun	2
Win32.Malware.Generic	1
Win32/Packed.Autoit.Gen	1
Worm.Win32.Ami	1
Total	117

Our implementations also rely heavily on BAP's processor model; errors in it will likely affect our results. Our experience has been that BAP is of high quality and actively improved. Also, BAP contains two largely independent x86 models; we have currently only used the default one, but repeating the experiments with the other model would be another way to improve confidence.

Limitations: There are two limitations of the proposed techniques. First, the policy is not fully automatically generated. Human expertise is still necessary to interpret the information flow results and devise special taint propagation and sanitization rules. In this respect, Godefroid and Taly presented algorithms to automatically generate the behavioral model of the arithmetic and logical x86 instructions as long as they follow certain templates [31]. The same techniques can also be used to help identify sanitization and propagation rules.

Second, trace based verification only provides a partial picture of whether the implementation is correct. On the other hand, since trace based verification recalculates the information flows and uses the ideal policy, it is possible for trace based verifiers, such as VITA, to be used to further refine the traces and reduce taints. The only requirement is that the original policy must not contain false negatives, since that might lead to an incomplete trace. As described previously, it is rare for taint analysis platforms to contain false negatives, and thus such a use is feasible.

VIII. Related Works

Our work is focused on context of generic dynamic taint analysis or dynamic information flow tracking; we do not make distinctions on how taint analysis is used. In this regard, Schwartz et al. provides an invaluable review of the different applications and the challenges [6].

Not all taint propagation policies are strictly based on the concept of noninterference. In Leakpoint, Clause et al. defined a taint propagation policy for pointer arithmetic [41] that is only loosely based on information flow. In empirical testing, Leakpoint was shown to be just as effective in identifying memory errors as Memcheck [32] which is based on information flow. Slowinska and Bos discussed the sources of false-positives and false-negatives in pointer tainting [38]. As mentioned before, memory arithmetic and operations is a weakness of VITA. In the end, since these applications do not adhere to noninterference, a different formal model must be defined to become the basis of further analysis. Once a formal model and the corresponding ideal policies are defined, and if that model is amendable for solving using SMT solvers, then the techniques presented in this paper can be reused in the new context.

Formal Analysis and Modeling: Schwartz et al. defined taint propagation semantics for the SIMPL language to reason about uses of taint analysis [6]. Tiwari et al. studied information flow in logic gates and implemented a taint tracker GLIFT [20]. Hu et al. presented both a formal model for information flow in logic gates as well as algorithms for generating precise taint propagation policies [42].

Newsome et al. measured the amount of *influence* on operands to increase precision [43]. They also relied on SMT solvers, STP [37] in their case, to solve constraints and obtain value range dependencies that is used to calculate influence. Their techniques can be used to quantify the amount of change while our technique, and VITA in particular, is used to identify information flow relationships.

Control Flow Dependencies: Researchers have used a combination of static and dynamic analysis to reduce false positives due to control flow dependency tracking. Bao et al. used static analysis to first identify "strict control dependence" relationships, (i.e. differences in the output values of variables in all branches are only due to static constants,) and then used it to reduce false positives [44]. Chang et al. first conducted general data-flow analysis on a program statically and used the results to direct information flow tracking at runtime [34]. In DTA++, Kang et al. used off-line analysis to identify "culprit" branches and limit control flow propagation to those branches [45]. This increased the precision of control flow propagation as compared to Dytan [7] which used simple control flow propagation rules. While this paper does not deal with control flow dependencies explicitly, we did highlight some of the challenges when dealing with instructions with internal control flow behavior.

There is also work on formal models for taint analysis that includes control flows. Volpano presented the formal model for a machine and a monitor that enforces a weak secrecy policy [46]. In terms of information flow, the weak

secrecy policy captures all flows except implicit control flows. Volpano continued to argue that "there is no monitor-enforced policy that is sound and complete for secrecy". Soundness and completeness are closely related to the false negative and false positives terms used in this paper and provides a theoretical limit to dynamic taint analysis. In short, dynamic taint analysis must have either false positives or false negatives.

IX. Conclusion

In this paper, we investigated a largely overlooked problem in dynamic taint analysis: correctness. More specifically, we focused on the soundness and precision of taint analysis. To study these two properties, we adopted the noninterference model to formally model taint propagation in a data-centric style and then used the model to define tainting rules and verified that all the rules are sound and most are precise. We also surveyed a list of publicly available taint analysis systems and found cases of unsoundness and extensive imprecision, which is surprising. Further, we implemented our tainting rules in a new taint analysis system called SPITA. With over 600,0000 test cases, we conducted per-trace verification and are confident in the correctness of our implementation. Compared with a widely used taint analysis system TEMU, SPITA is shown to be more precise and sound as well. Our evaluation on a large set of keyloggers further demonstrated the effectiveness of SPITA. All of these different test cases indicate that SPITA is indeed sound and mostly precise. The test cases were not exhaustive and the rules are not perfect though. Future work is needed to further improve the soundness and precision of dynamic taint analysis platforms like SPITA.

References

[1] J. Newsome and D. Song, "Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software," in *Proceedings of the 12th Annual Network and Distributed System Security Symposium (NDSS'05)*, February 2005.

[2] H. Yin, D. Song, E. Manuel, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the 14th ACM Conferences on Computer and Communication Security (CCS'07)*, October 2007.

[4] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of internet worms," in *In Proceedings of the Symposium on Systems and Operating Systems Principles (SOSP'05)*, 2005.

[5] V. Ganesh, T. Leek, and M. C. Rinard, "Taint-based directed whitebox fuzzing," in *International Conference on Software Engineering (ICSE)*, Vancouver, BC, Canada, May 2009, pp. 474–484.

[6] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Proceedings of the 2010 IEEE Symposium on Security and Privacy*, ser. SP '10. Washington, DC, USA: IEEE Computer Society, 2010, pp. 317–331. [Online]. Available: http://dx.doi.org/10.1109/SP.2010.26

[7] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*, ser. ISSTA '07. New York, NY, USA: ACM, 2007, pp. 196–206. [Online]. Available: http://doi.acm.org/10.1145/1273463.1273490

[8] F. Qin, C. Wang, Z. Li, H.-s. Kim, Y. Zhou, and Y. Wu, "Lift: A low-overhead practical information flow tracking system for detecting security attacks," in *Proceedings of the 39th Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 39. Washington, DC, USA: IEEE Computer Society, 2006, pp. 135–148. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2006.29

[9] V. P. Kemerlis, G. Portokalidis, K. Jee, and A. D. Keromytis, "libdft: practical dynamic data flow tracking for commodity systems," in *Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments*, ser. VEE '12. New York, NY, USA: ACM, 2012, pp. 121–132. [Online]. Available: http://doi.acm.org/10.1145/2151024.2151042

[10] E. Bosman, A. Slowinska, and H. Bos, "Minemu: The world's fastest taint tracker," in *Proceedings of RAID'11*, Menlo Park, CA, September 2011.

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

[12] "Bochs: The open source IA-32 emulation project," http://bochs.sourceforge.net/.

[13] J. Chow, B. Pfaff, T. Garfinkel, K. Christopher, and M. Rosenblum, "Understanding data lifetime via whole system simulation," in *USENIX Security Symposium*, San Diego, CA, USA, Aug. 2004, pp. 321–336.

[14] G. Portokalidis, A. Slowinska, and H. Bos, "Argos: an emulator for fingerprinting zero-day attacks," in *EuroSys 2006*, April 2006.

[15] 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, Jan 2010. [Online]. Available: http://www.eecs.berkeley.edu/Pubs/TechRpts/2010/EECS-2010-3.html

[16] L. K. Yan and H. Yin, "Droidscope: Seamlessly reconstructing os and dalvik semantic views for dynamic android malware analysis," in *Proceedings of the 21st USENIX Security Symposium*, ser. SEC'12. Berkeley, CA, USA: USENIX Association, 2012.

[17] N. Vachharajani, M. J. Bridges, J. Chang, R. Rangan, G. Ottoni, J. A. Blome, G. A. Reis, M. Vachharajani, and D. I. August, "Rifle: An architectural framework for user-centric information-flow security," in *Proceedings of the 37th annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 37. Washington, DC, USA: IEEE Computer Society, 2004, pp. 243–254. [Online]. Available: http://dx.doi.org/10.1109/MICRO.2004.31

[18] G. E. Suh, J. W. Lee, D. Zhang, and S. Devadas, "Secure program execution via dynamic information flow tracking," in *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, October 2004.

[19] M. Ozsoy, D. Ponomarev, N. Abu-Ghazaleh, and T. Suri, "Sift: a low-overhead dynamic information flow tracking architecture for smt processors," in *Proceedings of the 8th ACM International Conference on Computing Frontiers*, ser. CF '11. New York, NY, USA: ACM, 2011, pp. 37:1–37:11. [Online]. Available: http://doi.acm.org/10.1145/2016604.2016650

[20] M. Tiwari, H. M. Wassel, B. Mazloom, S. Mysore, F. T. Chong, and T. Sherwood, "Complete information flow tracking from the gates up," in *Proceedings of the 14th interna-*

tional conference on Architectural support for programming languages and operating systems, ser. ASPLOS '09. New York, NY, USA: ACM, 2009, pp. 109–120.

[21] J. Kong, C. C. Zou, and H. Zhou, "Improving software security via runtime instruction-level taint checking," in *Proceedings of the 1st workshop on Architectural and system support for improving software dependability*, ser. ASID '06. New York, NY, USA: ACM, 2006, pp. 18–24. [Online]. Available: http://doi.acm.org/10.1145/1181309.1181313

[22] S. Chen, J. Xu, N. Nakka, Z. Kalbarczyk, and R. Iyer, "Defeating memory corruption attacks via pointer taintedness detection," in *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*, june-1 july 2005, pp. 378 – 387.

[23] M. Dalton, H. Kannan, and C. Kozyrakis, "Raksha: a flexible information flow architecture for software security," in *Proceedings of the 34th annual international symposium on Computer architecture*, ser. ISCA '07. New York, NY, USA: ACM, 2007, pp. 482–493. [Online]. Available: http://doi.acm.org/10.1145/1250662.1250722

[24] L. De Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Proceedings of the Theory and practice of software, 14th international conference on Tools and algorithms for the construction and analysis of systems*, ser. TACAS'08/ETAPS'08. Berlin, Heidelberg: Springer-Verlag, 2008, pp. 337–340. [Online]. Available: http://dl.acm.org/citation.cfm?id=1792734.1792766

[25] J. Elgaard, N. Klarlund, and A. Møller, "MONA 1.x: new techniques for WS1S and WS2S," in *Proc. 10th International Conference on Computer-Aided Verification, CAV '98*, ser. LNCS, vol. 1427. Springer-Verlag, June/July 1998, pp. 516–520.

[26] L. Martignoni, S. McCamant, P. Poosankam, D. Song, and P. Maniatis, "Path-exploration lifting: hi-fi tests for lo-fi emulators," in *Proceedings of the seventeenth international conference on Architectural Support for Programming Languages and Operating Systems*, ser. ASPLOS '12. New York, NY, USA: ACM, 2012, pp. 337–348. [Online]. Available: http://doi.acm.org/10.1145/2150976.2151012

[27] J. A. Goguen and J. Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Computer Society Symposium on Research in Security and Privacy*. Oakland, CA: IEEE Computer Society Press, May 1982.

[28] "Valgrind: Project suggestions," http://valgrind.org/help/projects.html, July 2012.

[29] C. Barrett, A. Stump, and C. Tinelli, "The SMT-LIB Standard: Version 2.0," in *Proceedings of the 8th International Workshop on Satisfiability Modulo Theories (Edinburgh, UK)*, A. Gupta and D. Kroening, Eds., 2010.

[30] D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz, "BAP: A binary analysis platform," in *Proceedings of the 23rd International Conference on Computer Aided Verification*, ser. CAV'11. Berlin, Heidelberg: Springer-Verlag, 2011, pp. 463–469.

[31] P. Godefroid and A. Taly, "Automated synthesis of symbolic instruction encodings from i/o samples," in *Proceedings of the 33rd ACM SIGPLAN conference on Programming Language Design and Implementation*, ser. PLDI '12. New York, NY, USA: ACM, 2012, pp. 441–452. [Online]. Available: http://doi.acm.org/10.1145/2254064.2254116

[32] J. Seward and N. Nethercote, "Using valgrind to detect undefined value errors with bit-precision," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, ser. ATEC '05. Berkeley, CA, USA: USENIX Association, 2005, pp. 2–2. [Online]. Available: http://dl.acm.org/citation.cfm?id=1247360.1247362

[33] J. Ferrante, K. J. Ottenstein, and J. D. Warren, "The program dependence graph and its use in optimization," *ACM Trans. Program. Lang. Syst.*, vol. 9, no. 3, pp. 319–349, Jul. 1987. [Online]. Available: http://doi.acm.org/10.1145/24039.24041

[34] W. Chang, B. Streiff, and C. Lin, "Efficient and extensible security enforcement using dynamic data flow analysis," in *Proceedings of the 15th ACM conference on Computer and communications security*, ser. CCS '08. New York, NY, USA: ACM, 2008, pp. 39–50.

[35] D. Y. Zhu, J. Jung, D. Song, T. Kohno, and D. Wetherall, "Tainteraser: protecting sensitive data leaks using application-level taint tracking," *SIGOPS Oper. Syst. Rev.*, vol. 45, no. 1, pp. 142–154, Feb. 2011. [Online]. Available: http://doi.acm.org/10.1145/1945023.1945039

[36] J. L. Henning, "SPEC CPU2006 benchmark descriptions," *SIGARCH Computer Architecture News*, vol. 34, no. 4, pp. 1–17, 2006.

[37] V. Ganesh and D. L. Dill, "A decision procedure for bit-vectors and arrays," in *Proceedings of the 19th international conference on Computer aided verification*, ser. CAV'07. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 519–531.

[38] A. Slowinska and H. Bos, "Pointless tainting?: evaluating the practicality of pointer tainting," in *EuroSys '09*, April 2009.

[39] "TEMU: The BitBlaze dynamic analysis component," http://bitblaze.cs.berkeley.edu/temu.html.

[40] "BitBlaze: Binary analysis for COTS protection and malicious code defense," http://bitblaze.cs.berkeley.edu/.

[41] J. Clause and A. Orso, "Leakpoint: pinpointing the causes of memory leaks," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*, ser. ICSE '10. New York, NY, USA: ACM, 2010, pp. 515–524. [Online]. Available: http://doi.acm.org/10.1145/1806799.1806874

[42] W. Hu, J. Oberg, A. Irturk, M. Tiwari, T. Sherwood, D. Mu, and R. Kastner, "Theoretical fundamentals of gate level information flow tracking," *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, vol. 30, no. 8, pp. 1128 –1140, aug. 2011.

[43] J. Newsome, S. McCamant, and D. Song, "Measuring channel capacity to distinguish undue influence," in *Proceedings of the Fourth ACM SIGPLAN Workshop on Programming Languages and Analysis for Security (PLAS)*, Dublin, Ireland, Jun. 2009.

[44] T. Bao, Y. Zheng, Z. Lin, X. Zhang, and D. Xu, "Strict control dependence and its effect on dynamic information flow analyses," in *Proceedings of the 19th international symposium on Software testing and analysis*, ser. ISSTA '10. New York, NY, USA: ACM, 2010, pp. 13–24. [Online]. Available: http://doi.acm.org/10.1145/1831708.1831711

[45] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "DTA++: Dynamic taint analysis with targeted control-flow propagation," in *Proceedings of the 18th Annual Network and Distributed System Security Symposium*, San Diego, CA, Feb. 2011.

[46] D. M. Volpano, "Safety versus secrecy," in *Proceedings of the 6th International Symposium on Static Analysis*, ser. SAS '99. London, UK, UK: Springer-Verlag, 1999, pp. 303–311. [Online]. Available: http://dl.acm.org/citation.cfm?id=647168.718135