From trash to treasure: timing-sensitive garbage collection

Mathias V. Pedersen Aslan Askarov
Department of Computer Science
Aarhus University
{mvp,aslan}@cs.au.dk

***Abstract*—This paper studies information flows via timing channels in the presence of automatic memory management. We construct a series of example attacks that illustrate that garbage collectors form a shared resource that can be used to reliably leak sensitive information at a rate of up to 1 byte/sec on a contemporary general-purpose computer. The created channel is also observable across a network connection in a datacenter-like setting. We subsequently present a design of automatic memory management that is provably resilient against such attacks.**

I. Introduction

When a computer system allows third-party code to access sensitive information, it is necessary to ensure confidentiality of the sensitive information handled by the code. Language-based information flow control is a popular approach to solve this problem [33]. This approach uses programming language techniques to analyze information flows in the untrusted programs before and/or during the execution in a way that prevents potentially insecure code. The advantage of this approach is that it is allows fine-grained control, compared to coarse-grained systems approaches. The disadvantage is that source-code analysis is limited to flows that only have control graph representation, and that allows malicious code to still leak sensitive data using runtime side-channels such as CPU caches [28], schedulers [43], or programming-languages features such as lazy evaluation [8].

This paper studies another important aspect of program runtime – automatic memory management. We show that memory management represents a vulnerable shared resource through which an attacker can launder sensitive information. We present a series of simple attacks on modern runtimes, in particular Java sequential and parallel garbage collections and V8 default garbage collector, that illustrate the potential of the attack.

Attack model. We consider a threat model where an attacker-provided program operates on confidential information. The attacker observes the public input and output of the program, but does not observe either the secret input or the output. Furthermore, the attacker code is subjected to a number of syntactic and runtime checks that prevent it from directly leaking the secret input.

We assume that the attacker program consists of secret-dependent (high) and secret-independent (low) computations. High computations can access sensitive data, but cannot directly communicate with the attacker. Low computations may communicate with the attacker or affect the public output, but their execution cannot immediately depend on secrets.

Figure 1 illustrates the high-level idea behind the attacks. The attacks are designed so that the high computation influences the amount of allocated and reclaimable memory, which in its turn, influences the timing of allocations in the low computations, via the garbage collector. If there is no free space at the time of an allocation in the low computation, invocation of the garbage collector introduces observable delays. This delay is observed by two timing observations, before and after the the allocation in the low computation. The attacks do not rely on measuring the timing of the high computations – these in fact are considered to be secret-independent.

The low computation can do its timing observations using a number of ways. Our initial set of attacks uses the system clock. However, access to the local clock is not essential. Local time measurements can be substituted by external ones if the low computation has network access and can send dummy messages.

We further demonstrate that the observed channel can be amplified to leak arbitrary amount of information. Our experiments achieve the rate of nearly 1 byte/sec on a modern general-purpose laptop.

The attacks may not be surprising in hindsight, but their consequences are important. Automatic memory management is crucial in the implementation of modern object-oriented or functional programming languages. For strong-information security, a secure runtime is a must. Existing prototypes rely on source-to-source compilation [26, 34] or language-based monitoring [16, 35], yet they reuse commodity runtimes that are vulnerable to the types of attacks we describe here. We note that while there have been remarks in the literature about the danger of memory management in information flow settings [29], we are not aware of previously published attacks or proposals that focus on the timing channels through memory management.

To address the problem of leaks via garbage collection, we study a model of a programming language that uses abstract secure runtime and security types to enforce security. Our programming language includes a command for obtaining the current time, which allows us to cover a wide range of attacker models, because internal timing differences in a program can be converted into publicly observable output. Note the formal semantics of the language is designed so that it isolates leaks via garbage collection from other timing channels by using a non-

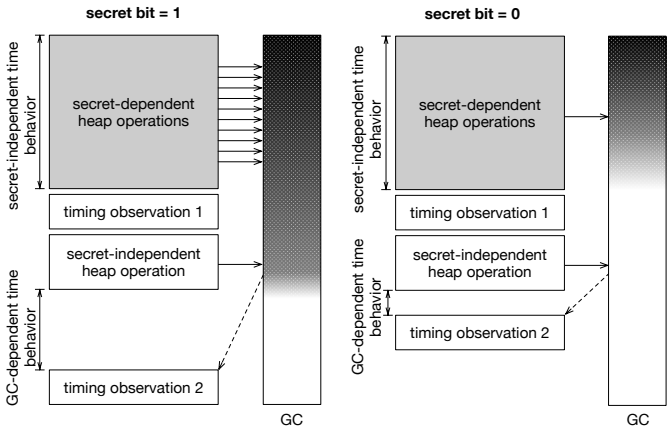


Fig. 1: Attack scenario – high computations are marked in gray; low computations are marked in white; both low and high computations share the memory management component (GC). Heap manipulation by the high process affects the state of the GC in a way that is later observable by the low computation.

standard prmitive at ℓ with bound e do c. This primitive pads the execution time of command c by the value of expression e.

We observe that garbage collection creates a bi-directional information flow channel. Securing garbage collection requires that high allocations cannot be collected in low computations and vice versa. We prove that the combination of these runtime restrictions with standard information flow type system is sufficient to close leaks via memory management.

In summary, the contributions of this paper are the following:

1) We develop the first amplifiable covert channel via automatic memory management. We show that the channel can be observed locally and over a network.
2) We observe that garbage collection creates a bi-directional information channel, which severely restricts the design space of securing automatic memory management. We present formal requirements that secure garbage collectors must satisfy.
3) We demonstrate that a secure garbage collector can be incorporated with traditional information flow analysis to provably establish a noninterference property.

The rest of the paper is structured as follows. Section II provides a background on timing channels and garbage collection. Section III explains our attacks and their results. Sections IV–V develop a formal language model for a small imperative language with allocatable arrays and garbage collection. Section VI designs a type system necessary for secure coordination with the runtime. Section VII studies the resulting security guarantees. Section VIII discusses the applicability of real-time garbage collectors in the light of the discovered attacks. Sections IX and X discuss the related work and conclude.

The proofs of the formal statements and the experimental artifacts can be found in the supplementary technical document [1].

II. Background

Programming languages that focus on information flow security statically reject programs that contain information flow violations, such as explicit and implicit flows. A common approach is to add security labels to types, corresponding to the confidentiality of the information stored in a variable of the given type. A natural consequence of security labels on types is the notion of a program counter label pc that determines the security context of the execution.

However, many such approaches ignore other potential sources of information leakage. A particularly dangerous source of information leaks is time.

Timing dependencies in programs may be direct or indirect [45].

```
if (h > 0) {
 /* long computation */
}
else {
 skip
}
```

An example of a direct timing dependency is the program on the left, where the decision to take one branch or the other depends on a confidential guard h. While direct timing attacks are difficult to close, they have control-flow representation, and thus are amenable to language-based mitigation [1, 45].

Indirect timing channel attacks are caused by interaction with the runtime system of the programming language, or the hardware upon which the program is running on. As such, they are much harder to close, because that requires careful interaction between the programming languages technology and the underlying runtime, often including the OS and the CPUs.

The focus of this paper is on indirect timing channel created by one aspect of language runtime system, specifically the automatic memory management.

We start with a brief overview of basic garbage collection concepts.

A. Garbage collection techniques

A garbage collector discovers which parts of the heap contain objects that will definitely not be accessed in the future, and then reclaims the memory occupied by these objects, allowing that memory to be reused.

Objects stored in the heap can point to other objects, and the heap objects thus form a directed graph. The root nodes of this object graph are the variables in the program, which provides "entry points" into the graph.

Determining whether an object will be accessed in the future is an undecidable problem [22], and thus garbage collection schemes conservatively approximate this property by assuming that every object reachable from the root nodes of the object graph will be accessed in the future.

The two garbage collection strategies attacked in this paper are mark-and-sweep collectors, and copy collectors.

[1] http://users-cs.au.dk/askarov/gc-timing/

a) Mark-and-sweep: A mark-and-sweep garbage collector operates in two phases: A marking phase, where all reachable objects are marked as "live", and a sweep phase, where unmarked objects in the heap are reclaimed. Note that the cost of a mark-and-sweep collection is the sum of the cost of marking, and the cost of sweeping. The cost of marking is linear in the size of the reachable objects, and the cost of sweeping is linear in the size of the entire heap.

One way to avoid having the cost be linear in the size of the entire heap is to use a copy collecting algorithm.

b) Copy collection: A copy collecting garbage collector partitions the heap in two partitions of equal size. These partitions are called *from-space* and *to-space*. An invariant of this algorithm is that at any point during the execution of the program, only the from-space is modified.

When the from-space partition is filled, the collector builds a copy of the object graph in the to-space partition. This is known as the evacuation phase. Afterwards all of the memory in the from-space is reclaimed, and the to-space becomes the new from-space, and vice versa.

c) Generational collection: Efficient garbage collectors avoid traversing the entire object graph by assuming the *weak generational hypothesis* which states that "most objects die young" [18, 37], meaning that newly allocated objects on the heap become unreachable fast.

The heap can then be partitioned in several partitions, known as generations. All allocations are initially stored in a small "young" generation, and a garbage collection invocation need only traverse this small partition. When an object survives a collection, it is moved to an "older" generation.

A minor collection is a collection that only traverses the object graph in the young generation, and a major collection is a collection that traverses both generations.

III. Attacking JVM and V8

This section presents two general amplifiable timing attack strategies that exploit the garbage collector in order to leak one bit of information. Both attacks work for two garbage collection strategies used by Java, as well as for the generational mark-sweep/mark-compact strategy used in V8.

The section also presents a technique for amplifying the one-bit leak to a general N-bit leak that works for all of the garbage collection strategies mentioned above.

A. High dependency in low context

This attack exploits the fact that during evacuation from from-space to to-space, the amount of bytes copied depends on the reachable nodes at the current point in the program. Thus, by creating a sufficiently large difference in reachable and unreachable nodes, the time required to perform a minor/major garbage collection becomes observable.

Figure 2 demonstrates the attack in Java. The example leaks whether $h > 0$ by observing the time difference caused by the allocation on line 15. If the value of *diff* is large then $h > 0$, and otherwise $h \leq 0$.

```
1  int[] a = new int[size1];
2  int[] b = null;
3  int[] c = null;
4  int[] d = null;
5  if (h > 0) {
6    b = new int[size1];
7    d = a;
8  }
9  else {
10   c = new int[size1];
11   b = a;
12 }
13 c = null;
14 long before = System.nanoTime();
15 int[] x = new int[size2];
16 long after = System.nanoTime();
17 long diff = after - before;
```

Fig. 2: Program leaking one bit of information based on evacuation time.

The attack works as follows. Suppose that constants `size1` and `size2` are chosen so that the following constraints hold, where $collectionThreshold$ is an experimentally obtained constant that triggers garbage collection, and S is the number of bytes required to represent an integer in Java.

- $2 \cdot S \cdot \texttt{size1} \leq collectionThreshold$
- $S \cdot (2 \cdot \texttt{size1} + \texttt{size2}) \geq collectionThreshold$

Line 1 allocates a new array, and keeps a reference to that array in the variable a. If $h > 0$ we allocate a new array on line 6 and store a reference to this array in the variable b. On the other hand, if $h \leq 0$ we allocate a new array and store the reference in the variable c, which will become unreachable as soon as we reach line 13. Note that on line 11 we keep a reference to the array allocated on line 1, meaning that a and b point to the same array. Now assume the allocation on line 15 invokes the garbage collector. If $h > 0$ there will be two distinct arrays that need to be copied from from-space to to-space, meaning that $2 \cdot \texttt{size1}$ integers will be copied. However, if $h \leq 0$ then the only array which is reachable, and thus should be copied, is the array allocated on line 1. Thus we only copy `size1`. This difference in the number of bytes that should be copied creates an observable difference in timing.

B. Low modification in high context

The previous program demonstrated how the evacuation of sensitive information in public contexts can lead to a covert channel. The next attack shows how garbage collection of public information in a sensitive context also leads to a covert channel. Consider the program in Figure 3 and suppose that constants `size1` and `size2` are chosen so that the following constraints hold.

- $S \cdot \texttt{size2} \leq collectionThreshold$
- $S \cdot (\texttt{size1} + \texttt{size2}) \geq collectionThreshold$

If $h > 0$ the allocation on line 2 partially fills up the heap, so that the allocation on line 7 triggers a garbage collection, and thus the value of diff is large. However, if $h > 0$ is false

```
1  if(h > 0) {
2    int[] b = new int[size1];
3    b = null;
4  }
5
6  long before = System.nanoTime();
7  int[] c = new int[size2];
8  long after = System.nanoTime();
9  long diff = after - before;
```

Fig. 3: Program leaking one bit of information based on the presence of a garbage collection.

then no garbage collection occurs on line 7, as the size of the memory does not exceed the implementation's threshold for garbage collection. This results in a small value for diff.

C. Amplifying the attacks

We now amplify the leakage of the attacks described in Section III-A and III-B. This leads to an attack that leaks the value of a 32 bit integer. To illustrate the technique we describe how the attack from Section III-A can be extended.

First, note that the attacks cannot be extended naively by repeating the algorithm for each bit. To see why, assume this approach is taken and that we enter the first iteration with 0% of the available memory having been allocated. We would like to keep this 0% allocated memory as a loop invariant, so that we can repeat the attack for each bit.

Next, we might allocate memory equal to 75% of the available memory, which we turn into garbage by removing any referencing to the allocated memory. We then perform a "leaky allocation" of some amount of memory which will cause a garbage collection to occur (ie. we request strictly more than 25% of the available memory, such that we in total have requested strictly more than 100% of the available memory, forcing a GC to occur). This collects the 75% of available memory, but it also gives us a non-zero block of memory in return, meaning that we enter the next iteration of the loop with a non-zero percent of the available memory having been allocated. Thus the invariant has been broken.

Thus we must modify the attack in several ways. The resulting attack is shown in Figure 4.

First, we repeat the attack N (where $N \in [10, 20]$ is sufficient) times and measure each trial run. If the allocation duration is larger than some threshold, we store the time required to perform the allocation in the array times. By filtering out allocations with short allocation times we filter out iterations that does not lead to an invocation of the garbage collector.

Second, instead of allocating one array when filling up the memory, we allocate K arrays, where K is usually less than 10. This increases the evacuation time since more memory will need to be copied. Larger values of K will lead to a greater timing difference between a zero bit and a one bit, at the cost of a greater allocation time. Thus there is a time/precision trade off.

```
1  long[] times = new long[N];
2  int guess = 0;
3
4  for(int bit = 31; bit >= 0; --bit) {
5    for(int i = 0; i < N; ++i) {
6      int[][] a = new int[K][size];
7      int[][] b = null;
8      int[][] c = null;
9      int[][] d = null;
10     if (((secret >> bit) & 1) > 0) {
11       b = new int[K][size];
12       d = a;
13     }
14     else {
15       c = new int[K][size];
16       b = a;
17     }
18     c = null;
19     long before = System.nanoTime();
20     int[] c = new int[size2];
21     long after = System.nanoTime();
22     if(after - before > threshold) {
23       times[i] = after - before;
24     }
25     else {
26       times[i] = 0;
27     }
28   }
29
30   long sum = 0;
31   int numOfGCs = 0;
32   for(int i = 0; i < times.length; ++i) {
33     long t = times[i];
34     if(t != 0) {
35       sum += t;
36       ++numOfGCs;
37     }
38   }
39   if(numOfGCs == 0) {
40     ++bit;
41     continue;
42   }
43   if(sum / numOfGCs > DELTA) {
44     guess += Math.pow(2, bit);
45   }
46 }
```

Fig. 4: Program leaking 32 bits of information based on evacuation time.

Third, we compute the average allocation time for each of the trial runs which invoked the garbage collector. If the average garbage collection time is above some chosen DELTA we conclude that the current bit is one, and zero otherwise.

Finally, in case no garbage collection occurs we retry the current iteration. Note that the probability of invoking a collection in subsequent tries is increased because more memory has been allocated from the previous tries of the same iteration, meaning that several tries of the same iteration is rare.

D. Results

The blue plot of Figure 5a shows the output of running the program described in Figure 4 on the secret input 5342121 with the serial garbage collection strategy used by Java when invoked with the parameter -XX:+UseSerialGC. Similarly the red part of the figure shows the output of the same program, with modified constants, on the same secret input with the parallel collection strategy used when invoking Java with the parameter -XX:+UseParallelGC.[2]

Figure 5b shows the output obtained by running a similar attack on the V8 JavaScript engine using Node.js. This attack follows the same pattern as the attack in Figure 4, and has therefore been omitted.

The experiment results are gathered from a machine with the following specs: Intel(R) Core(TM) i7-5557U CPU @ 3.10GHz, Memory: 8 GB. NodeJS version 6.2.0. Java version "1.8.0_77", Java(TM) SE Runtime Environment (build 1.8.0_77-b03), Java HotSpot(TM) 64-Bit Server VM (build 25.77-b03, mixed mode).

All figures show a clear distinction in garbage collection time consumption in the aftermath of processing a one bit, and processing a zero bit.

a) Observations over network: The timing observations in these attacks are of sufficient magnitude to be observed over an internal network, e.g., in a datacenter-like setting, with ping latency of 0.5ms. Figure 5c shows the timing observed by a client communicating with a server over 25 trials, where each trial consists of the following operations:

1) first, the server sends a ping to the client
2) then, the server performs an allocation similar to the allocation on line 15 in Figure 2
3) finally, the server sends another ping to the client.

Figure 5c thus shows the difference in the delay between the two pings sent by the server during step (1) and (3). The red bars show the delay when roughly half as much memory should be garbage collected.

b) Rate: By measuring the execution time of the program in Figure 4 over 25 iterations we calculate the rate of the channel obtained. This yields a channel rate of 0.98 bytes per second.

In the sections to follow, we construct a formalism for studying these attacks. We first introduce a standard imperative language using a small-step semantics, with a few technical deviations to facilitate an isolated study of garbage collection. We extend the semantics to incorporate garbage collection transitions, and prove that our garbage collection semantics satisfies functional correctness. Finally we add a standard type system for information flow which, when combined with the semantics of Section IV-B, implies resilience against the presented attacks.

IV. Language

This section presents a design of a small imperative programming language with automatic memory management. The key element of the design is that careful combination of the guarantees obtained via typing and the runtime constraints on the memory management eliminate timing leaks via garbage collection.

a) Syntax: Figure 6 describes the syntax of our language. It is a standard imperative language [41, 33, 45] extended with heap allocated arrays and the corresponding getters and setters, a command for obtaining the current time, and the two security-related constructs, as explained below. For expressions, n ranges over the set of integers $\mathbb{Z}$ and x, y, z range over variables. Finally, op ranges over binary integer operations. A special expression null corresponds to the only memory location representable at the source level.

b) Non-standard features: The runtime of the language has an explicit notion of time that may be observed programmatically using command $x := \mathsf{time}()$. This particular design choice has the advantage that it provides a powerful attacker model without complicating the formal setup, e.g., introducing intermediate outputs. This includes a network attacker who observes timing of the network communication, as well an attacker providing untrusted code with access to a clock, which may in general be needed for functionality.

Our formal semantics (cf. Section IV-A) models the time using simple instruction counting – operational steps in the computation. This simplification is sufficient for our purposes of expressing the fundamental constraints necessary to eliminate timing leaks via memory management. Naturally, a realistic implementation would need to soundly relate the operational steps with the wall-clock, but that is outside of the scope of the current work.

In order to secure the behavior of the garbage collector at runtime, the language contains a notion of the *runtime program counter level*. The program counter is manipulated via two security-relevant constructs. Command $\mathsf{at}\ \ell\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c$ raises the program counter level, while command $\mathsf{restore}\ \ell\ \mathsf{when}\ n$ lowers it. Note that $\mathsf{restore}$ command never appears in the program source, but is explicitly inserted in the operational semantics to restore the level of the program counter level after an $\mathsf{at}\ \ell\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c$ command [23].

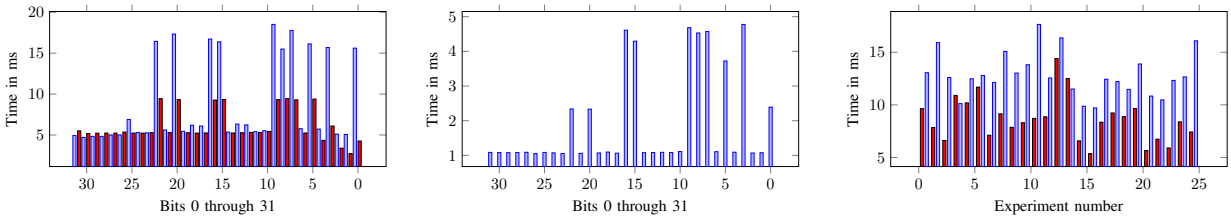
Another aspect of at and $\mathsf{restore}$ commands is that they implement lightweight predictive mitigation [45, 4, 44] of direct timing channels, i.e., channels that have control-flow representation, such as secret conditionals. An execution of statement $\mathsf{at}\ \ell\ \mathsf{with\ bound}\ n\ \mathsf{do}\ c$ is padded to take exactly n steps. The execution is blocked if c takes more than n steps. This particular design aspect provides clear containment of direct timing channels, which in its turn allows isolated study of the leaks via memory management.

A. Semantic enivronments

Our formal semantics partitions the program memory into memory environment m, which models variables that are typically allocated on stack, and the heap environment h.

a) Values, variables, and locations: A value in the language is either an integer or a location in the heap, including

[2]The appendix contains a link to supplementary material with a VM containing all of our examples.



(a) Running leaky program with a blue: serial (red: parallel) collection strategy on secret input 5342121

(b) Running the GC strategy of V8 on secret input 5342121

(c) Measuring ping delay between computers over a network connection.

Fig. 5: Experimental results for network ping delay and running leaky programs on sample input $5342121 = (00000000\ 01010001\ 10000011\ 10101001)_2$

$$
\begin{aligned}
e &::= n \mid \mathsf{null} \mid x \mid e\ \mathsf{op}\ e \\
c &::= \mathsf{skip} \mid x := e \mid c; c \mid \mathsf{if}\ e\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \mathsf{do}\ c \\
&\quad \mid x := \mathsf{new}_\ell(e, e) \mid x := y[e] \mid x[e] := e \\
&\quad \mid x := \mathsf{time}() \mid \mathsf{at}\ \ell\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c \\
&\quad \mid \boxed{\mathsf{restore}\ \ell\ \mathsf{when}\ n}
\end{aligned}
$$

Fig. 6: Syntax of the language. The boxed command is not part of the surface syntax.

null. For our purposes it is sufficient to leave heap locations abstract, and our model simply assumes a set Loc of abstract heap locations that may be distinguished from each other. We further assume that the set of locations is disjoint from the set of integer values. Let v range over values Val, and denote the set of all variables as Var.

b) Memory and heap environments: Memory environment is a partial function $m : \mathsf{Var} \to \mathsf{Val}$. For convenience, we also use the set notation, and write $(x, v) \in m$ when $m(x) = v$.

For heap environment, an important characteristic of our model is that allocations on the heap are tagged with security levels. This is necessary for constraining collecting behavior, as explained later in this section.

We define a heap as a partial function $h : \mathsf{Loc} \to (\mathbb{N} \to \mathsf{Val}) \times \mathcal{L}$ from heap locations to pairs consisting of *lookup functions* and security levels. A lookup function is a partial function from the integer offsets to values stored in the heap. We write $(loc, \mu, \ell) \in h$ when $h(loc) = (\mu, \ell)$, where loc is the abstract location, μ is the lookup function, and ℓ is the security level. We refer to ℓ as the *heap level* of loc. When heap h can be split into two disjoint heaps h_1 and h_2, we write $h = h_1 \uplus h_2$.

c) Maximum heap size: Our model considers both unbounded and bounded heaps. The semantics is parametrized with the function that returns the maximum size of the heap $\mathsf{mx} : \mathcal{L} \to \mathbb{N} \cup \{\infty\}$ that returns the maximum amount of heap memory available for allocation at that security level. Unless explicitly specified, the formal results apply to both bounded and unbounded heaps.

When the current size of the heap h at a particular level ℓ needs to be compared against the maximum available space, we compute the current size using function $\mathsf{size}_\ell(h) \triangleq \sum_{(loc,\mu,\ell)\in h} |\mathsf{dom}(\mu)|$.

B. Semantics w/o collection

We start by introducing the semantics of the language w/o garbage collection. Section V presents the semantics of garbage collection.

Semantic configurations have the form $\langle c, pc, m, h, t\rangle$, where c is the current program, pc is the runtime program counter level, m and h are the memory and the heap respectively, and t is the time counter. Terminal configurations are marked by the dedicated stop command. The semantics is a combination of a standard big-step evaluation relation for expressions, and a small-step transition relation for commands.

1) Expressions: Figure 7 presents the evaluation relation $\langle x, m\rangle \Downarrow v$ for expression semantics. Note how this relation only includes standard memory; all heap-related operations are modeled as commands.

2) Commands: Figures 8 and 9 present the semantics of commands in the absence of garbage collection. None of the standard commands touch the heap or the program counter level; moreover they all take one computation step. This semantics is given by rule (S-LIFT-STANDARD) in Figure 9 together with the standard transition relation $\langle c, m\rangle \to \langle c', m'\rangle$, defined in Figure 8.

The remaining of the rules in Figure 9 present the transition relation for the and non-standard commands.

Rule (S-TIME) updates variable x with the current value of the time counter.

Rule (S-NEW) models the allocation in partition ℓ. The amount of allocated memory is computed by evaluating expression e. The command extends the heap with the new lookup function μ, and updates the variable in the memory with the value of the new location. The lookup function μ uses the default value computed by evaluating expression e_{init}. The rule has two notable constrains. First, the location must be fresh, expressed by the premise $loc \notin \mathsf{dom}(h)$. Second, there must be enough available space in the heap at partition ℓ for the allocation, which is expressed by the premise

$$\frac{}{\langle n, m\rangle \Downarrow n} \qquad \frac{}{\langle \mathsf{null}, m\rangle \Downarrow \mathsf{null}} \qquad \frac{m(x) = v}{\langle x, m\rangle \Downarrow v}$$

$$\frac{\langle e_i, m\rangle \Downarrow v_i \quad i = 1, 2 \qquad v_1 \ \mathsf{op}\ v_2 = v}{\langle e_1 \ \mathsf{op}\ e_2, m\rangle \Downarrow v}$$

Fig. 7: Semantics of expressions

$$\frac{}{\langle \mathsf{skip}, m\rangle \to \langle \mathsf{stop}, m\rangle} \qquad \frac{\langle e, m\rangle \Downarrow v}{\langle x := e, m\rangle \to \langle \mathsf{stop}, m[x \mapsto v]\rangle}$$

$$\frac{\langle c_1, m\rangle \to \langle c_1', m'\rangle \qquad c_1' \neq \mathsf{stop}}{\langle c_1; c_2, m\rangle \to \langle c_1'; c_2, m'\rangle} \qquad \frac{\langle c_1, m\rangle \to \langle \mathsf{stop}, m'\rangle}{\langle c_1; c_2, m\rangle \to \langle c_2, m'\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \qquad n \neq 0 \Rightarrow i = 1 \qquad n = 0 \Rightarrow i = 2}{\langle \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2, m\rangle \to \langle c_i, m\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \qquad n \neq 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle \to \langle c; \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle}$$

$$\frac{\langle e, m\rangle \Downarrow n \qquad n = 0}{\langle \mathsf{while}\ e\ \mathsf{do}\ c, m\rangle \to \langle \mathsf{stop}, m\rangle}$$

Fig. 8: Semantics of standard commands

$\mathsf{size}_\ell(h') \leq \mathsf{mx}(\ell)$, where h' refers to the heap updated after the allocation.

Rule (S-SET) updates the array at a specified index. This is expressed as in-place update of the lookup function. Rule (S-GET) retrieves the value stored at a particular index in an array; the result is stored in the memory.

Rule (S-AT) updates the program counter label. Additionally, this rule computes the time n the at command is expected to consume. This rule inserts a restore command that restores the program counter, and the expected time $t + n$ by which the restore should be reached.

Rule (S-RESTORE-PROGRESS) restores the program counter label and continues with the execution of the body of the restore command only if the current time matches the expected time specified in the restore command. Rule (S-RESTORE-WAIT) skips until the current time matches the argument of the restore command. Note that if the body of at happens to take more time than expected, the semantics blocks the execution.

V. SEMANTICS FOR SECURE GARBAGE COLLECTION

The previous section defined the program semantics w/o garbage collection transitions, i.e., the size of the heap would monotonically increase throughout the execution. This section defines the collection semantics that specifies how garbage collection is allowed to affect the heap.

The main insight is that the runtime program counter label constrains which parts of the heap can be collected and when. When the runtime program counter is low, only low parts of the heap can be collected; when the runtime program counter is high, only high parts of the heap can be collected. While, on first sight, this isolation-like constrain may appear unnecessarily strong in an information-flow setting, it is necessary, because the garbage collection represents a bi-directional information flow channel. We explain this using two simple examples inspired by our experiments from Section III.

A. Motivating security restrictions on GC

This section presents two examples that motivate our restrictions on garbage collection semantics. Each of the examples demonstrates the danger of collecting parts of the memory that do not match the current program counter level. Note that while the examples are written in the style that follows our typing discipline of Section VI, the typing is not required here.

1) Implicit flows when collecting L in H: Consider program below, where we assume that N and M are constants, and v is picked sufficiently pessimistically to bound the execution of the at command.

```
1  // new array of size N with
2  // default element value 0
3  y := new_L (N, 0);
4  y := null;
5  // the y-array can now be reclaimed
6  at H with bound v do {
7    if h > 0 {
8        // new array of size M that
9        // requires GC
10       x := new_H(M,0)
11   } else {
12       skip
13   }
14 }
15 t1 := time();
16 // GC time depends on
17 // if y-array has been
18 // collected earlier
19 y  := new_L (N,0);
20 t2 := time();
21 low := t2 - t1
```

Note how the high conditional is guarded by the at command that ensures that the execution of the conditional takes v steps. This means that the value of t_1 does not depend on which branch of the conditional is taken.

However, if semantics of garbage collection allows low parts to be collected inside at, say before executing the allocation on Line 10, then $t_2 - t_1$ is likely to be small. This motivates that garbage collection should not collect low allocations when the program counter level is high.

2) Implicit flows when collecting H in L: This example shows that collecting high allocations when the program counter is low is also dangerous. Suppose we are given constants M, N, and v as described earlier, and consider program below.

```
1  // new array of size M with
2  // default element value 0
3  x := new_H (M, 0);
4  at H with bound v do {
```

S-LIFT-STANDARD

$$\frac{\langle c, m\rangle \to \langle c', m'\rangle}{\langle c, pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle c', pc, m', h, t+1\rangle}$$

S-TIME

$$\frac{}{\langle x := \mathsf{time}(), pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{stop}, pc, m[x \mapsto t], h, t+1\rangle}$$

S-NEW

$$\frac{\langle e, m\rangle \Downarrow n \qquad loc \notin \mathsf{dom}(h) \qquad \mathsf{size}_\ell(h') \le \mathsf{mx}(\ell) \qquad \langle e_{init}, m\rangle \Downarrow v \qquad h' = h[loc \mapsto (\mu, \ell)] \qquad \mu(x) = \begin{cases} v & 0 \le x < n \\ \text{undef} & \text{otherwise} \end{cases}}{\langle x := \mathsf{new}_\ell(e, e_{init}), pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{stop}, pc, m[x \mapsto loc], h', t+1\rangle}$$

S-SET

$$\frac{\langle e_1, m\rangle \Downarrow n \qquad \langle e_2, m\rangle \Downarrow v \qquad 0 \le n < |\mathsf{dom}(\mu)| \qquad loc = m(x) \qquad (\mu, \ell) = h(loc) \qquad \mu'(x) = \begin{cases} v & x = n \\ \mu(x) & \text{otherwise} \end{cases}}{\langle x[e_1] := e_2, pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{stop}, pc, m, h[loc \mapsto (\mu', \ell)], t+1\rangle}$$

S-GET

$$\frac{\langle e, m\rangle \Downarrow n \qquad loc = m(y) \qquad \mu = h(loc) \qquad v = \mu(n)}{\langle x := y[e], pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{stop}, pc, m[x \mapsto v], h, t+1\rangle}$$

S-AT

$$\frac{\langle e, m\rangle \Downarrow n \qquad t' = t + n}{\langle \mathsf{at}\ \ell\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c, pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle c; \mathsf{restore}\ pc\ \mathsf{when}\ t', \ell, m, h, t+1\rangle}$$

S-RESTORE-PROGRESS

$$\frac{t = t'}{\langle \mathsf{restore}\ \ell\ \mathsf{when}\ t', pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{stop}, \ell, m, h, t+1\rangle}$$

S-RESTORE-WAIT

$$\frac{t < t'}{\langle \mathsf{restore}\ \ell\ \mathsf{when}\ t', pc, m, h, t\rangle \overset{\mathsf{mx}}{\to} \langle \mathsf{restore}\ \ell\ \mathsf{when}\ t', pc, m, h, t+1\rangle}$$

Fig. 9: Semantics of heap and non-standard commands

```
5    if h > 0 {
6        x := null
7        // array x can now be reclaimed
8    } else {
9        skip
10   }
11 }
12 t1 := time();
13 // GC time depends on whether
14 // array x is reclaimable
15 y  := new_L (N,0);
16 t2 := time();
17 low := t2 - t1
```

As before, the timing of the high conditional is protected with an at command. Consider allocation on Line 15 that may trigger garbage collection. If semantics of the GC allows collecting high allocation in the low program counter, the amount of time that the collector spends here will depend on whether the array x can be reclaimed, affecting the value of t_2. This motivates that garbage collection should not collect high allocations when the program counter is low.

B. Formal semantics for garbage collection

Using the above examples as guideline, we now formulate the formal semantics for secure garbage collection. The rule for garbage collection is given by transition relation $\langle c, pc, m, h, t\rangle \dashrightarrow \langle c, pc, m, h', t'\rangle$ that relates two configurations before and after collection. Collection does not update the current command or memory or the program counter level, but updates the heap and consumes some time.

Figure 10 presents the collection rule formally. To explain the rule, we introduce the auxiliary concepts that it uses.

1) Abstract collection relation: The amount of time consumed by the collection is in general implementation-specific. We require the implementation to provide an interface for collecting specific *parts* of the heap. We model this by an abstract relation $h \rightsquigarrow^m_\delta h'$, where h is a subheap, and where h' is the result of collection in h that takes time δ, given memory m. An important constraint that we place on the $\rightsquigarrow$ relation is that if two subheaps and starting memories are *isomorphic*, then it must take the same amount of time to

GC-COLLECT

$$\frac{\mathsf{reach}(m, h_1 \uplus h_2) \cap \mathsf{dom}(h_2) = \emptyset \qquad {h_2}^{\neq pc} = \emptyset \qquad h_1 = {h_1}^{\neq pc} \uplus {h_1}^{=pc} \qquad {h_1}^{=pc} \uplus h_2 \rightsquigarrow_\delta^m {h_1}^{=pc}}{\langle c, pc, m, h_1 \uplus h_2, t\rangle \dashrightarrow \langle c, pc, m, h_1, t + \delta\rangle}$$

Fig. 10: Reduction rule for garbage collection

collect in them. To formally express this, we introduce the notion of substitutions. This notion of substitution is closely related to the one by Banerjee and Naumann [6]. As we will see, the substitution is also used later in the paper when defining our GC requirement.

Definition 1. *(Substitution) A substitution* $\phi : \mathsf{Val} \to \mathsf{Val}$ *is a mapping such that*

1) ϕ *is identity on integers:*

$$\forall n \;.\; \phi(n) = n.$$

2) ϕ *is injective on locations:*

$$\forall loc\ loc' \;.\; \phi(loc) = \phi(loc') \Rightarrow loc = loc'.$$

3) ϕ *maps locations to locations:*

$$\forall loc \in \mathsf{Loc} \;.\; \phi(loc) \in \mathsf{Loc}.$$

Given a substitution ϕ, we write $\phi(m) = \{(x, \phi(v)) \mid (x, v) \in m\}$ and $\phi(h) = \{(\phi(loc), \phi \circ \mu, \ell) \mid (loc, \mu, \ell) \in h\}$. For the remaining parts of the paper it is assumed that substitutions are bijective. That is,

$$\forall v, v' \;.\; \phi(v) = v' \Leftrightarrow \phi^{-1}(v') = v.$$

We write $h \cong_\phi w$ (resp. $(m, h) \cong_\phi (s, w)$) when $\phi(h) = w$ (resp. $(\phi(m), \phi(h)) = (s, w)$) and $h \cong w$ (resp. $(m, h) \cong_\phi (s, w)$) when there exists a bijection ϕ such that $h \cong_\phi w$ (resp. $(\phi(m), \phi(h)) = (s, w)$). Two heaps are then isomorphic when $h \cong w$.

Using the notion of substitution, we formulate our assumption on the abstract collection relation.

Assumption (Abstract collection)**.** *Consider two memories* m *and* s *and two heaps* h *and* w *and a substitution* ϕ *such that* $(s, w) = (\phi(m), \phi(h))$*. Then* $h \rightsquigarrow_\delta^m h'$ *implies* $w \rightsquigarrow_\delta^s w'$

If two heaps are isomorphic they are equal up to renaming of locations, and the specific names of locations should not affect the behaviour of garbage collection.

2) Heap partitioning based on a level: Given a heap h and a level ℓ, write $h^{=\ell}$ for the heap that includes all allocations tagged with security level ℓ:

$$h^{=\ell} \triangleq \{(x, \mu, \ell') \in h \mid \ell' = \ell\}.$$

and similarly, define the complement partition as

$$h^{\neq \ell} \triangleq \{(x, \mu, \ell') \in h \mid \ell' \neq \ell\}.$$

Given a level ℓ, any heap h can be decomposed into a disjoint union of its partition and its complement: $h = h^{=\ell} \uplus h^{\neq \ell}$.

3) Reachable locations: Since deciding whether or not a location will be accessed in the future is undecidable [22], we follow real world implementations of garbage collection schemes, and treat a variable as live if it is reachable from the current set of variables in the program. The set of reachable values is then the values that are pointed to by variables in the memory, or by following a chain of reachable locations on the heap.

Definition 2 (Reachable locations)**.** *Given a memory* m *and a heap* h*, the set of* reachable locations $\mathsf{reach}(m, h) \subseteq \mathsf{Loc}$ *is the smallest set such that*

1) all locations in memory m *are reachable:*

$$\mathsf{cod}(m) \cap \mathsf{Loc} \subseteq \mathsf{reach}(m, h)$$

2) if $loc \in \mathsf{reach}(m, h)$ *and* $h(loc) = (\mu, \ell)$ *then locations that the allocation* μ *points to are reachable:*

$$\mathsf{cod}(\mu) \cap \mathsf{Loc} \subseteq \mathsf{reach}(m, h)$$

Note that reach is monotonic, as expressed by the following lemma.

Lemma 1 (Monotonicity of heap reachability)**.** *Given two heaps* h *and* w*, if* $h \subseteq w$ *then* $\mathsf{reach}(m, h) \subseteq \mathsf{reach}(m, w)$.

With these definitions at hand, let us examine the garbage collection rule in Figure 10. The rule is defined when the program heap is split in two disjoint heaps h_1 and h_2, where h_2 is collected after the transition. That it is functionally safe to collect h_2 is ensured by the first premise of the rule that stipulates that no location in h_2 is reachable from the current configuration. The remaining premises induce security restrictions on the collection. We restrict h_2 to only contain allocations that are exactly at the level of the program counter level pc – this is expressed by the requirement ${h_2}^{\neq pc} = \emptyset$, which could alternatively be stated as ${h_2}^{=pc} = h_2$. The idea is to constrain collections at a specific level only when the program counter matches that level. The rule further splits the non-collectable heap h_1 into two parts, based on the security level: the pc-part ${h_1}^{=pc}$ and its complement ${h_1}^{\neq pc}$. Only the pc-part of the heap is used when invoking the abstract collector.

C. Functional correctness

The remaining part of this section shows that our garbage collection strategy is functionally correct. That is, the collector never claims memory that is accessed in the future. We start with a formal definition of dangling pointer-freedom.

Definition 3 (Dangling pointer-freedom)**.** *Given a memory* m *and heap* h*, say that* (m, h) *is* free of dangling pointers *when*

1) all locations in the memory point to a valid location in the heap:

$$\mathsf{cod}(m) \cap \mathsf{Loc} \subseteq \mathsf{dom}(h)$$

2) pointers within heap are valid:

$$\forall (\mu, \ell) \in \mathsf{cod}(h) \;.\; \mathsf{cod}(\mu) \cap \mathsf{Loc} \subseteq \mathsf{dom}(h)$$

Informally, if m and h do not contain dangling pointers, then extending the heap does not increase reachability.

Lemma 2. *Given memory m and h such that (m, h) is free of dangling pointers then for all heaps $w \supseteq h$, it holds*

$$\mathsf{reach}(m, h) = \mathsf{reach}(m, w)$$

For functional correctness, we show that adding the garbage collection rule in Figure 10 to the set of possible transitions does not modify the memory, nor the reachable part of the heap.

There is a technical challenge to overcome. The specific location allocated in the heap, and stored in the memory, depends on the current size of the heap (cf. the semantics in Figure 9). Thus by adding a garbage collector, which reduces the size of the heap, the locations allocated and stored in the memory will be different from the locations allocated without first reducing the size of the heap.

We therefore prove that the reachable parts of the heaps will be equal up to renaming of locations. We use the notion of substitution introduced earlier in Section V-B1 to relate pairs of memories and heaps that have isomorphic reachable parts but may have different amounts of garbage.

Definition 4 (Matching up to garbage)**.** *Consider two pairs of memory and heaps (m, h) and (s, w). Say that (s, w)* matches *(m, h)* up to garbage via substitution *ϕ, written $(m, h) \simeq_\phi (s, w)$ if $\mathsf{reach}(\phi(m), \phi(h)) = \mathsf{reach}(s, w)$.*

We write $(m, h) \simeq (s, w)$ if there exists ϕ s.t. $(m, h) \simeq_\phi (s, w)$.

The following lemma states that the semantics does not depend on garbage.

Let $\omega(\ell) = \infty$ denote the constant ∞ function. The relation $\overset{\omega}{\rightarrow}$ defines the semantics in an abstract setting with unbounded available memory.

Lemma 3 (Garbage independence in unbound heaps)**.** *For all heaps h and w such that $w \supseteq h$, and (m, h) and (m, w) are free of dangling pointers it holds that if*

$$\langle c, pc, m, h, t\rangle \overset{\omega}{\rightarrow} \langle c', pc', m', h', t'\rangle$$

then $\langle c, pc, m, w, t\rangle \overset{\omega}{\rightarrow} \langle c', pc', s', w', t'\rangle$ and $(m', h') \simeq (s', w')$.

The next lemma states that garbage collection only collects garbage from the heap. That is, if garbage collection collects some portion of the heap, then succeeding transitions do not depend on the portion of the heap that has been collected.

Lemma 4 (Garbage only)**.** *If (m, h) is free of dangling pointers and*

$$\langle c, pc, m, h, t\rangle \overset{\mathsf{mx}}{\rightarrow} \langle c', pc', m', h', t'\rangle$$

then for all s, w such that (s, w) is free of dangling pointers, and $(s, w) \simeq (m, h)$ we have

$$\langle c, pc, s, w, t\rangle \overset{\mathsf{mx}}{\rightarrow} \langle c', pc', s', w', t'\rangle.$$

and $(s', w') \simeq (m', h')$.

We can now state functional correctness of the garbage collection scheme. Intuitively, the reachable heap does not change when interleaving the reductions with garbage collections. This is expressed as a pair of theorems, in the style of the work by Morrisett et al. [22]. The first theorem states that running the GC followed by a regular transition is comparable to running a regular transition. The second one states that running a regular transition is comparable to running the GC followed by a regular transition.

Note that in the statements of the theorems below the time component of the configurations is reset. This ensures that the result of running command $x := \mathsf{time}()$ is the same. Furthermore, the first theorem additionally qualifies the semantics to be unbounded in maximum available size. This is needed because otherwise the execution may run out of available heap.

Theorem 1 (Functional correctness for unbound heaps with time reset)**.** *Consider memory m and heaps h and w such that (m, h) and (m, w) are free of dangling pointers. If $\langle c, pc, m, h, t\rangle \dashrightarrow \langle c, pc, m, w, t''\rangle$ and $\langle c, pc, m, w, t\rangle \overset{\omega}{\rightarrow} \langle c', pc', m', w', t'\rangle$ then $\langle c, pc, m, h, t\rangle \overset{\omega}{\rightarrow} \langle c', pc', s', h', t'\rangle$ and $(m', w') \simeq (s', h')$.*

Theorem 2 (Functional correctness with time reset)**.** *Consider memory m and heaps h such that (m, h) is free of dangling pointers. If $\langle c, pc, m, h, t\rangle \overset{\mathsf{mx}}{\rightarrow} \langle c', pc', m', h', t'\rangle$ then $\langle c, pc, m, h, t\rangle \dashrightarrow \langle c, pc, m, w, t''\rangle$ and $\langle c, pc, m, w, t\rangle \overset{\mathsf{mx}}{\rightarrow} \langle c', pc', s', w', t'\rangle$ and $(s', w') \simeq (m', h')$.*

D. GC or normal steps

As a final element in this section, we define a top-level *GC or normal step* as a transition function that nondeterministically interleaves normal and collection steps.

$$\textsc{Normal-Step}\quad \frac{\langle c, pc, m, h, t\rangle \overset{\mathsf{mx}}{\rightarrow} \langle c', pc', m', h', t'\rangle}{\langle c, pc, m, h, t\rangle \xrightarrow{\mathsf{mx}} \langle c', pc', m', h', t'\rangle}$$

$$\textsc{GC-Step}\quad \frac{\langle c, pc, m, h, t\rangle \dashrightarrow \langle c, pc, m, h', t'\rangle}{\langle c, pc, m, h, t\rangle \xrightarrow{\mathsf{mx}} \langle c, pc, m, h', t'\rangle}$$

We use this top-level relation in studying security properties of our programs in Section VII.

VI. Type system

In addition to the secure garbage collection described in the previous section, our enforcement mechanism additionally relies on a typing discipline. The typing discipline is mostly standard for an imperative security-typed language with arrays [10, 39, 25, 31, 9], with a few minor technical deviations that we explain below. In particular, the type system ensures not only confidentiality, but also integrity by viewing values that depend on time commands as tainted. This restricts the extent to which the result of the time command affects the control flow or the heap shape of the program.

$$\text{WF-INT}\quad \frac{}{\ell \vdash_{\mathsf{wf}} \mathbf{int}\,\xi} \qquad \text{WF-ARRAY}\quad \frac{\ell \sqsubseteq \ell_p \qquad \ell_{ref} \sqsubseteq \ell_p \qquad \ell_p \vdash_{\mathsf{wf}} \tau}{\ell \vdash_{\mathsf{wf}} \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)}$$

Fig. 11: Type well-formedness

A. Time taint and generalized security levels

We introduce a *time lattice*; a two-point lattice with the elements $\circ$ and $\bullet$. Here, $\circ$ corresponds to to *untainted* values, and $\bullet$ corresponds to *tainted* values. We let ι range over elements of this lattice, and define an ordering $\sqsubseteq^{\mathsf{t}}$ such that for all $\iota \in \{\circ, \bullet\}$ it holds that $\iota \sqsubseteq^{\mathsf{t}} \iota$, and $\circ \sqsubseteq^{\mathsf{t}} \bullet$. We define the corresponding least upper bound operator as $\sqcup^{\mathsf{t}}$.

A *generalized security level* ξ is a combination of both a confidentiality level ℓ and a taint level ι. With this, we have the following grammar for the security levels in the type system.

$$\iota ::= \circ \mid \bullet$$
$$\xi ::= (\ell, \iota)$$

We lift lattice operations to generalized security levels, and denote the resulting ordering and least upper bound operations as $\preceq$ and $\curlyvee$, respectively.

Types are given by the following grammar

$$\sigma ::= \mathbf{int} \mid \mathbf{array}_{\ell}[\tau]$$
$$\tau ::= \sigma\,\xi$$

Here, τ is a security annotated type that consist of a base type with a security level. Base types σ are either integers or arrays of some type τ that specify the confidentiality level of the partition where the array lives.

Given a base type σ and security levels ξ_1, ξ_2, define the operator for raising of the type $\sigma\,\xi_1$ to level ξ_2 as

$$(\sigma\,\xi_1)^{\xi_2} \triangleq \sigma\,(\xi_1 \curlyvee \xi_2)$$

The lattice ordering $\preceq$ induces a subtyping relation on the types

$$\frac{\xi_1 \preceq \xi_2 \qquad \sigma_1 = \sigma_2}{\sigma_1\,\xi_1 <: \sigma_2\,\xi_2}$$

Note that invariance in the base types, even if the base type is $\mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)$, is required because the arrays are mutable [24].

B. Well-formedness of reference types

Figure 11 presents well-formedness conditions of types w.r.t. references. The security level on the left-hand side of the turnstile is a lower bound on the heap level that can store values of type τ. A type τ is well-formed when $\bot \vdash_{\mathsf{wf}} \tau$. These rules prevent creation of references from the high partitions into the low ones, and are later lifted to define well-formedness of configurations.

$$\text{T-INT}\quad \frac{}{\Gamma \vdash n : \mathbf{int}\,\xi} \qquad \text{T-NULL}\quad \frac{\bot \vdash_{\mathsf{wf}} \mathbf{array}_{\ell}[\tau](\ell_{ref}, \circ)}{\Gamma \vdash \mathsf{null} : \mathbf{array}_{\ell}[\tau]\,(\ell_{ref}, \circ)}$$

$$\text{T-VAR}\quad \frac{}{\Gamma \vdash x : \Gamma(x)} \qquad \text{T-OP}\quad \frac{\Gamma \vdash e_i : \mathbf{int}\,\xi_i \quad i = 1, 2}{\Gamma \vdash e_1\ \mathsf{op}\ e_2 : \mathbf{int}\,(\xi_1 \curlyvee \xi_2)}$$

Fig. 12: Typing rules for expressions

C. Typing rules

We assume a memory typing environment Γ that maps variable names to types. In the remaining of the paper, we require that the memory typing environments are well-formed w.r.t. all types defined in it. The typing judgment for expressions has form $\Gamma \vdash e : \tau$. Figures 12 presents the typing rules for expressions.

The typing judgment for commands has form $\Gamma, pc \vdash c$. Figure 14 present the typing rules for commands, where pc is the static program counter level. The rule (T-SKIP) is trivial. Rule (T-ASSIGN) is standard in how it prevents both implicit and explicit information flows using the program counter level. The rule (T-TIME) requires the assigned variable to be marked as tainted, and is otherwise similar to assignment in its treatment of implicit flows. Rule (T-IF) is slightly non-standard. First, it prevents branching on high data if the pc is low. Note that the rule does not raise the level of the program counter label in the branches. Instead, high conditionals must occur syntactically in a scope where the pc-level is explicitly raised using `at` command. Second, branching on high values is allowed only if the value is not tainted by time commands. This is visualized in Figure 13. Rule (T-WHILE) imposes a similar restriction, and is otherwise standard.

Rule (T-NEW) requires that both pc-level and the level of the expression that determines the size of the array flow to the variable that stores the reference. This prevents the size of a high array from depending on low values. The level ℓ on the command that is interpreted by the allocation semantic (cf. Rule (S-New) in Section V) must be as restrictive as the level ℓ_2 of the reference.

Rule (T-SET) requires that both the pc-level and the expression used for indexing flow to the level of the array reference. This prevents indexing into a low array using high expressions. It also requires the type of the right-hand side expression to flow to the type of the array on the left, taking implicit flows via pc-level into account.

Rule (T-GET) is similar. It requires that the pc-level and the level of the index expression flows to the level of the array reference, and rules out both explicit and implicit flows in the assignment.

Finally, rules (T-AT) raises the level of pc, Furthermore, an explicit time bound is provided for this command, which allows the programmer to control the time consumed by commands when the pc is high.

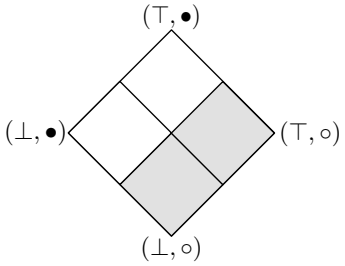


Fig. 13: Lattice for confidentiality and integrity. Branching and heap manipulation is allowed on values whose type is in the gray area.

$$\text{T-SKIP}\quad \frac{}{\Gamma, pc \vdash \mathsf{skip}} \qquad \text{T-ASSIGN}\quad \frac{\Gamma \vdash e : \sigma\,\xi \qquad \sigma\,(\xi \curlyvee (pc, \circ)) <: \Gamma(x)}{\Gamma, pc \vdash x := e}$$

$$\text{T-IF}\quad \frac{\Gamma \vdash e : \mathbf{int}\,(\ell, \circ) \qquad \ell \sqsubseteq pc \qquad \Gamma, pc \vdash c_i \quad i = 1, 2}{\Gamma, pc \vdash \mathsf{if}\ e\ \mathsf{then}\ c_1\ \mathsf{else}\ c_2}$$

$$\text{T-WHILE}\quad \frac{\Gamma \vdash e : \mathbf{int}\,(\ell, \circ) \qquad \ell \sqsubseteq pc \qquad \Gamma, pc \vdash c}{\Gamma, pc \vdash \mathsf{while}\ e\ \mathsf{do}\ c}$$

$$\text{T-SEQ}\quad \frac{\Gamma, pc \vdash c_i \quad i = 1, 2}{\Gamma, pc \vdash c_1; c_2}$$

$$\text{T-NEW}\quad \frac{\Gamma \vdash e_{size} : \mathbf{int}\,(\ell_{size}, \circ) \qquad \Gamma \vdash e_{init} : \tau \qquad \Gamma(x) = \mathbf{array}_\ell[\tau]\,(\ell_x, \circ) \qquad pc \sqcup \ell_{size} \sqsubseteq \ell_x}{\Gamma, pc \vdash x := \mathsf{new}_\ell(e_{size}, e_{init})}$$

$$\text{T-SET}\quad \frac{\Gamma \vdash e_{idx} : \mathbf{int}\,\xi_{idx} \qquad \Gamma \vdash e : \sigma\,\xi \qquad \Gamma(x) = \mathbf{array}_\ell[\tau]\,\xi_x \qquad \xi_{idx} \curlyvee (pc, \circ) \preceq \xi_x \qquad \sigma\,(\xi_x \curlyvee \xi) <: \tau}{\Gamma, pc \vdash x[e_{idx}] := e}$$

$$\text{T-GET}\quad \frac{\Gamma \vdash e_{idx} : \mathbf{int}\,\xi_{idx} \qquad (pc, \circ) \curlyvee \xi_{idx} \preceq \xi_y \qquad \Gamma(y) = \mathbf{array}_\ell[\sigma\,\xi]\,\xi_y \qquad \sigma\,(\xi \curlyvee \xi_y) <: \Gamma(x)}{\Gamma, pc \vdash x := y[e]}$$

$$\text{T-TIME}\quad \frac{\Gamma(x) = \mathbf{int}\,(\ell, \bullet) \qquad pc \sqsubseteq \ell}{\Gamma, pc \vdash x := \mathsf{time}()}$$

$$\text{T-AT}\quad \frac{\Gamma \vdash e : \mathbf{int}\,(\ell', \circ) \qquad \ell' \sqsubseteq pc \sqsubseteq \ell \qquad \Gamma, \ell \vdash c}{\Gamma, pc \vdash \mathsf{at}\ \ell\ \mathsf{with\ bound}\ e\ \mathsf{do}\ c}$$

Fig. 14: Typing rules for the surface language commands

Because the information flow constraints imposed by the type system are standard, they can also be enforced using dynamic or hybrid monitors [32, 5].

Note that the type system permits time measurements in both low and high context; this directly models attacker capability to make internal timing measurements (cf. Section III); weaker attacker models, i.e., the ones where attacker does not have access to system clock but only to network messages, can be addressed in a similar manner.

1) Properties of the type-system: The type system ensures two important properties. To state these, we need a heap typing environment [38] that maps allocated locations to types.

Definition 5 (Heap typing environment)**.** *A* heap typing *is a partial function* $\Sigma : \mathsf{Loc} \to \tau$ *that maps heap locations to their types.*

The intuition for Σ is that given a location loc, allocated by a command $x := \mathsf{new}_\ell(e, e_{init})$, where $\Gamma \vdash e_{init} : \tau$, we have $\Sigma(loc) = \tau$. Similarly to memory typing environments, we assume that types defined by the heap typing environment are well-formed.

We can now state the first property, which we split into two sub-properties: one for typing environments Γ, and one for heap typing environments Σ.

First, the typing environment Γ gives us an adequate view of the heap level of locations. More specifically, if a variable x points to a location loc, then the heap level of loc equals to the partition level specified by $\Gamma(x)$.

Similarly, we can state this property for a heap typing environment Σ. Let loc be a location. Then $\Sigma(loc)$ records the type of the "content" stored at loc. So, if $\Sigma(loc) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)$ it means that the content of location loc has a heap level of ℓ_p. That is, following loc twice leads to a heap allocation with a heap level equal to ℓ_p.

The second property is that the type system prevents creating pointers from high heap levels into the low heap levels. This property is important because a pointer from a high heap level to a low heap level would allow modifying low heap level pointers in a high program context.

Definition 6. *Given a memory* m*, a heap* h*, an typing environment* Γ *and a heap typing environment* Σ *we say that* (m, h) *is well-formed wrt.* (Γ, Σ) *if*

1a) For all variables x *s.t.* $\Gamma(x) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)$*,* $m(x) = loc$*, and* $h(loc) = (\ell, \mu)$*, it holds that* $\ell = \ell_p$*.*

1b) For all locations loc s.t. $\Sigma(loc) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)$*,* $h(loc) = (\ell, \mu)$*,* $\mu(n) = loc'$*, and* $h(loc') = (\ell', \nu)$ *it holds that* $\ell' = \ell_p$*.*

2) If $h(loc_1) = (\ell_1, \mu_1)$*, and* $\mu_1(n) = loc_2$ *for some* $n \in \mathbb{N}$*, and* $h(loc_2) = (\ell_2, \mu_2)$ *then* $\ell_1 \sqsubseteq \ell_2$*.*

We define well-formed configurations to be configurations in which the command is well-typed, the memory and heaps are heap level bound and are free of dangling pointers. Finally we also capture the intuition about the relation between Γ and Σ: That Σ contains the type of the "content" of a location, wheres Γ contains the type of the location.

Definition 7 (Well-formed configuration)**.** *Given a configuration $\langle c, pc, m, h, t\rangle$, a typing environment Γ and a heap typing environment Σ, say that the configuration is* well-formed *w.r.t. Γ, Σ, if*

1) $c \neq \mathsf{stop} \Rightarrow \Gamma, pc \vdash c$
2) (m, h) *is free of dangling pointers.*
3) (m, h) *is well-formed wrt.* (Γ, Σ).
4) *If* $\Gamma(x) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ)$ *and* $m(x) = loc$ *then* $\Sigma(loc) = \tau$.

By a standard proof of preservation [42] the semantics can be shown to preserve the well-formedness of configurations.

VII. Security guarantees

This section presents the security guarantees obtained by combining the properties of partitioned allocation semantics (Section IV), constrained garbage collection (Section V), and security types (Section VI).

For garbage collection in isolation we obtain a property of timing-sensitive noninterference.

For programs in general, the semantic security property obtained in this section is *termination-insensitive timing-sensitive noninterference*. While this may appear unorthodox, given the usual expectation that timing-sensitivity implies termination-sensitivity, we believe it makes sense in our setting, where the attacker has access to the internal clock of the computation, yet there are many ways via which the program may diverge. The sources of divergence may be infinite loops – that we allow, heap exhaustion – also possible in our semantics, or other fatal errors that we do not currently model. In that light, even though we allow termination channels in this work, it remains a channel that cannot be efficiently magnified [3]. Note however that termination-insensitivity is not a fundamental restriction in our work. Because our type system is relatively standard, it should be possible to apply orthogonal techniques [21] to obtain termination or progress-sensitive security.

Our notion of noninterference is parametrized over the heap size, because of the parametrization of the semantics.

A. ℓ-equivalence

In order to formalize our security conditions, we introduce ℓ-equivalence [45] for memories and heaps.

We define the set of low-reachable locations, written $\mathsf{reach}_\ell(\Gamma, \Sigma, m, h)$, as the set of locations loc satisfying the predicate $\mathsf{reach}_\ell(loc, \Gamma, \Sigma, m, h)$, which is specified in Figure 15. Intuitively, this is the set of locations reachable by following only pointers with a low confidentiality level according to the memory and heap typing environments. An important property of low reachability is that, when a location maps to a low heap level, then the low reachability of that location coincides with its reachability as per Definition 2.

Lemma 5 (Adequacy of low reachability)**.** *Let (m, h) be well-formed wrt. (Γ, Σ) and let ℓ be a security level. If $h(loc) = (\ell, \mu)$ and $\ell' \sqsubseteq \ell$ then $loc \in \mathsf{reach}_\ell(\Gamma, \Sigma, m, h)$ if and only if $loc \in \mathsf{reach}(m, h)$.*

$$\frac{\Gamma(x) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ) \qquad \ell_{ref} \sqsubseteq \ell \qquad m(x) = loc}{\mathsf{reach}_\ell(loc, \Gamma, \Sigma, m, h)}$$

$$\frac{\mathsf{reach}_\ell(loc, \Gamma, \Sigma, m, h) \qquad \Sigma(loc) = \mathbf{array}_{\ell_p}[\tau]\,(\ell_{ref}, \circ) \qquad \ell_{ref} \sqsubseteq \ell \qquad h(loc) = (\ell', \mu) \qquad \mu(n) = loc'}{\mathsf{reach}_\ell(loc', \Gamma, \Sigma, m, h)}$$

Fig. 15: Low-reachability

Lemma 5 will be a crucial part of the proof of garbage collection noninterference in Section VII-B.

a) Memory low-equivalence: We first consider a definition of low-equivalence for memories. This relation is induced by Γ.

$$m \overset{\Gamma,\phi}{\sim}_\ell s \triangleq \forall x \in \mathsf{dom}(\Gamma)\ .\ m(x) \overset{\Gamma(x)}{=}_{\phi,\ell} s(x)$$

Whenever Γ is clear from the context, we omit it for clarity.

b) Memory and heap low-equivalence: As we wish to we reason about garbage collection, which identifies and removes unreachable locations, reasoning about locations that are low reachable is not sufficient to prove the desired non-interference results. Thus, we define the set of *low locations* as not only the low reachable ones, but also the locations which has a low heap level. This is captured in the following definition.

Definition 8 (Low locations)**.** *Given security typing Γ, heap typing environment Σ, memory m, and heap h, the set $\mathsf{Low}_\ell(\Gamma, \Sigma, m, h)$ is the smallest set such that*

1) *Low reachable locations are contained in the set.*

$$\mathsf{reach}_\ell(\Gamma, \Sigma, m, h) \subseteq \mathsf{Low}_\ell(\Gamma, \Sigma, m, h)$$

2) *Locations with a low heap level are contained in the set.*

$$\forall loc\ .\ h(loc) = (\ell', \mu) \wedge \ell' \sqsubseteq \ell \Rightarrow loc \in \mathsf{Low}_\ell(\Gamma, \Sigma, m, h).$$

We can now define heap low equivalence similarly to memory low-equivalence. As we will often be relating two environments, we use the abbreviation $\Sigma_{1,2}$ to mean a set of heap typing environments Σ_1 and Σ_2. The same abbreviation is used for memories and heaps.

Crucial for the non-interference is the idea that if two locations are related by a bijection ϕ, then one location is reachable if and only if the other location is reachable. This is captured in the following definition.

$$\begin{aligned}&\mathsf{reach\text{-}iff}^{\tau}_{\phi,\ell}(\Sigma_{1,2}, m, h, s, w) \triangleq \\ &\quad \forall loc_1\ loc_2\ .\ \phi(loc_1) = loc_2 \Rightarrow \\ &\qquad loc_1 \in \mathsf{reach}_\ell(\Gamma, \Sigma_1, m, h) \Leftrightarrow loc_2 \in \mathsf{reach}_\ell(\Gamma, \Sigma_2, s, w)\end{aligned}$$

The final relation specified by State-Low-Eq, written $(m, h) \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\phi,\ell} (s, w)$, is parametrized by the level ℓ of the low-equivalence (typically low), the typing environment Γ, the two heap typing environments Σ_1 and Σ_2, and the substitution ϕ that witnesses the isomorphism between the heaps h and w. This is the main relation of interest during execution, as this

$$\textsc{Same-Type-Int}\quad \frac{}{n_1 \equiv n_2} \qquad \textsc{Same-Type-Loc}\quad \frac{}{loc_1 \equiv loc_2}$$

$$\textsc{Val-Low-Eq}\quad \frac{\tau = \sigma\,(\ell', \iota) \qquad v \equiv u \qquad \ell' \sqsubseteq \ell \Rightarrow \phi(v) = u}{v \stackrel{\tau}{=}_{\phi,\ell} u}$$

$$\textsc{Heap-Loc-Eq-Non-Reach}\quad \frac{loc_i \notin \mathsf{reach}(m_i, h_i) \text{ for } i = 1,2 \qquad \mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2) \qquad h_i(loc_i) = (\ell', \mu_i) \text{ for } i = 1,2}{\mathsf{heap\text{-}loc\text{-}eq}^{\tau}_{\phi,\ell}(m_{1,2}, h_{1,2}, loc_1, loc_2)}$$

$$\textsc{Heap-Loc-Eq-Reach}\quad \frac{loc_i \in \mathsf{reach}(m_i, h_i) \text{ for } i = 1,2 \qquad \mathsf{dom}(\mu_1) = \mathsf{dom}(\mu_2) \qquad h_i(loc_i) = (\ell', \mu_i) \text{ for } i = 1,2 \qquad \forall n\ .\ \mu_1(n) \stackrel{\tau}{=}_{\phi,\ell} \mu_2(n)}{\mathsf{heap\text{-}loc\text{-}eq}^{\tau}_{\phi,\ell}(m_{1,2}, h_{1,2}, loc_1, loc_2)}$$

$$\textsc{State-Low-Eq}\quad \frac{\begin{array}{l} m \overset{\Gamma,\phi}{\sim}_{\ell} s \qquad \mathsf{reach\text{-}iff}^{\tau}_{\phi,\ell}(\Sigma_{1,2}, m, h, s, w) \\ \forall loc_1\ loc_2\ .\ \phi(loc_1) = loc_2 \wedge \\ \quad \Sigma_1(loc_1) = \tau\ \wedge\ \Sigma_2(loc_2) = \tau \wedge \\ \quad loc_1 \in \mathsf{Low}_{\ell}(\Gamma, \Sigma_1, m, h) \wedge \\ \quad loc_2 \in \mathsf{Low}_{\ell}(\Gamma, \Sigma_2, s, w) \\ \quad \Rightarrow \mathsf{heap\text{-}loc\text{-}eq}^{\tau}_{\phi,\ell}(m, s, h, w, loc_1, loc_2) \end{array}}{(m,h) \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\phi,\ell} (s,w)}$$

Fig. 16: Low-equivalence on memories and heaps

what relates the parts of the environments that the attacker can observe. It relates the low reachability of the two environments, and specifies that the memories are related, and finally the last relation expresses that if two low locations are related by ϕ and have data labeled at a common type τ then they must have the same heap level, and if the locations are reachable then the values associated with this location (i.e. the codomain of μ_1 and μ_2 in HEAP-LOC-EQ-REACH) have to be equivalent at type τ. When Γ and $\Sigma_{1,2}$ are irrelevant, we write $(m,h) \approx_{\phi,\ell} (s,w)$ for $(m,h) \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\phi,\ell} (s,w)$.

B. Noninterference for garbage collection

Using the definition of memory and heap low-equivalence we can formulate the noninterference result for the garbage collector. Our definition of noninterference for garbage collection is possibilistic [20] in its nature. Intuitively, it states that for a GC-transition that takes some time there is a GC-transition that takes as much time and yields a low-equivalent resulting heap and memory.

Theorem 3 (Garbage collection noninterference)**.** *Assume typing environment Γ, level ℓ and heap typing environments Σ_1 and Σ_2. Consider two well-formed configurations $\langle c_1, pc, m, h, t\rangle$ and $\langle c_2, pc, s, w, g\rangle$ wrt. Γ, Σ_1, and Γ, Σ, a substitution ϕ such that $(m,h) \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\phi,\ell} (s,w)$. Assume $pc \sqsubseteq \ell$. If $\langle c_1, pc, m, h, t\rangle \dashrightarrow \langle c_1, pc, m, h', t + \delta\rangle$ then there is w' and ψ such that*

$$\langle c_2, pc, s, w, g\rangle \dashrightarrow \langle c_2, pc, s, w', g + \delta\rangle$$

and $(m,h') \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\psi,\ell} (s,w')$.

Proof sketch. Unfolding we have $h = h^{=pc} \uplus h^{\neq pc} \uplus h^{gc}$, where h^{gc} is the subheap being collected. Pick

- $w^{=pc} = \phi(h^{=pc})$,
- $w^{gc} = \phi(h^{gc})$.

By definition no location in h^{gc} is reachable, and so they are not low-reachable either. Then since $\mathsf{reach\text{-}iff}^{\tau}_{\phi,\ell}(\Sigma_{1,2}, m, h, s, w)$ no location in $w^{gc} = \phi(h^{gc})$ must be low-reachable and since all locations in w^{gc} have heap level $pc \sqsubseteq \ell_{\mathsf{adv}}$ it follows that no location in w^{gc} is reachable, meaning that it is safe to GC this part of the heap.

By definition of GC we have $h^{=pc} \uplus h^{gc} \rightsquigarrow^{m}_{\delta} h^{=pc}$ and so $w^{=pc} \uplus w^{gc} \rightsquigarrow^{s}_{\delta} w^{=pc}$ by the GC assumption from Section V. □

A property of the state low equivalence relation is that garbage collection when $pc \not\sqsubseteq \ell_{\mathsf{adv}}$ results in a state which is low equivalent to the state before garbage collection.

Lemma 6 (High garbage collection)**.** *Assume typing environment Γ and level ℓ. Consider configuration $\langle c, pc, m, h, t\rangle$ and assume $pc \not\sqsubseteq \ell$. If $\langle c, pc, m, h, t\rangle \dashrightarrow \langle c, pc, m, h', t'\rangle$ then $(m,h) \approx_{\mathsf{id},\ell} (s,h')$.*

C. Noninterference for programs

Formal attacker observations. To simplify the technical presentation we assume that the secrets in the computation are all stored in the initial memory.

We present our noninterference condition using the notion of attacker knowledge [2, 12]. The attacker knowledge is the set of possible memories that are are consistent with the memory after a sequence of program transitions. We assume that programs start with empty heap $\emptyset$ and an initially low pc level $\bot$.

Definition 9 (Attacker knowledge at level ℓ)**.** *Given program c, initial and final memories m and m', final heap h', security level ℓ and maximum heap size function* mx*, define attacker knowledge as*

$$k^{\mathsf{mx}}_{\ell}(c, m, m', h') \triangleq \{s \mid m \overset{\Gamma,\mathsf{id}}{\sim}_{\ell} s \wedge \langle c, \bot, s, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^{*} \langle \mathsf{stop}, pc', s', w', t'\rangle \wedge \exists \phi, \Sigma_{1,2}\ .\ (m', h') \overset{\Gamma,\Sigma_{1,2}}{\approx}_{\phi,\ell} (s', w')\}$$

Note that the larger attacker knowledge set corresponds to attacker obtaining less information. Smaller knowledge sets correspond to more precise information. Singleton knowledge set means the attacker knows the exact initial memory with which the execution started.

Definition 10 (Set of terminating memories)**.** *Given a program c we define $\mathbb{M}_\ell^{\mathsf{mx}}(c, m)$ as the set of initial ℓ-equivalent memories that lead to a terminating configuration when the heap is bounded by* mx*.*

$$\mathbb{M}_\ell^{\mathsf{mx}}(c, m) = \{s \mid m \overset{\Gamma,\mathsf{id}}{\sim}_\ell s \wedge \langle c, \bot, s, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^* \langle \mathsf{stop}, pc', m', h', t'\rangle\}$$

Using attacker knowledge and the set of initial memories we can define the noninterference policy [15]. Intuitively, a program satisfies noninterference if any memory and heap produced by a terminating sequence of program steps does not exclude any possible initial memory.

Definition 11 (Termination-Insensitive Noninterference at ℓ for heap size mx)**.** *Given a heap bounding function* mx*, a program c satisfies* mx-noninterference up to level ℓ *if for all initial memories m such that*

$$\langle c, \bot, m, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^* \langle \mathsf{stop}, pc', m', h', t'\rangle$$

implies

$$k_\ell^{\mathsf{mx}}(c, m, m', h') \supseteq \mathbb{M}_\ell^{\mathsf{mx}}(c, m).$$

Intuitively, this definition says that a program c satisfies termination-insensitive noninterference when, given two terminating executions of the same program with low equivalent initial memories m and s and final memories m' and s', it is possible to construct a terminating execution of c starting at initial memory s which results in a final memory s'' such that s'' is low equivalent to s'.

Theorem 4 (Soundness of the enforcement)**.** *Given a program c, if $\Gamma, pc \vdash c$ then c satisfies noninterference for unbounded semantics for all levels ℓ.*

Proof sketch. Given two terminating executions of program c:

$$\langle c, \bot, m, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^* \langle \mathsf{stop}, pc_1, m', h', t'\rangle$$
$$\langle c, \bot, s, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^* \langle \mathsf{stop}, pc_2, s^{*\prime}, w^{*\prime}, g^{*\prime}\rangle$$

(call these executions A and B resp.) our goal is to construct an alternative run (call this execution C)

$$\langle c, \bot, s, \emptyset, 0\rangle \xrightarrow{\mathsf{mx}}{}^* \langle \mathsf{stop}, pc_1, s', w', t'\rangle$$

where (m', h') is $\approx$-equivalent to (s', w').

To show this, we start by defining an auxiliary "bridge" relation on pairs of configurations. The relation records that starting from some configuration cfg_1, the execution "steps over" a number of intermediate steps, that do not modify the low parts of the memory or the heap, or terminates, resulting in configuration cfg_2. Each of the A and B executions can be broken down into a sequence of consecutive "bridging" steps. We construct the execution C one bridge-step at a time, starting from the initial configurations. The key invariant used in the proof is that bridging configurations in C execution are low-equivalent with the respective configurations in A and "taint"-equivalent with the respective configurations in B. Our workhorse bridge noninterference shows that whenever a pair of related configurations in A and B can take a bridge step, it is possible to construct a matching bridge step in C that "mimics the timing behavior" of A. i.e., it generates the same events and takes the same execution time as A. This is proved by induction on the number of the intermediate steps, followed by an induction over the structure of command c. In order to show the termination of the high commands constructed in C run, we observe that the taint equivalence of B and C configurations implies that they agree on the control flow, and one can further construct high GC steps in C to match the high GC steps in B.

□

VIII. Connections to real time garbage collection

Wadler [40] defined the term *real time garbage collection* as any garbage collection system guaranteeing that the execution is not suspended for long periods of time. Many such collectors have been presented previously [30, 17, 36, 13], and they are a crucial part of building real time systems in managed languages [7].

As the main goal of real time garbage collectors is to reduce the amount of time the garbage collector suspends the program execution, such collectors could be seen as an effective mitigation against the attacks presented in Section III.

While we have not performed experiments against existing real time garbage collectors at this point, we expect that real time garbage collectors are not sufficient for mitigating these attacks. To see this, consider the second program from Section V-A that illustrates the danger of collecting H in L. An "eager" real-time garbage collector may manifest the behavior that this example warned against. Additionally, if a practical implementation of real-time garbage collection algorithm occasionally stops the world for collection, this allows attacks similar to the first example in Section V-A.

IX. Related work

Conceptually, this work fits into the framework introduced by Zhang et al. [45], where the interaction between the language semantics and the underlying abstract runtime happens via security labels. The work by Zhang et al. does not however consider automatic memory management.

Many modern programming language features can be used to create timing channels, and programming language designers must secure the entire execution stack. Buiras and Russo [8] show that a programming language with lazy evaluation leak information because of sharing. Buiras and Russo [8] breaks the information flow control of the Haskell library *LIO* by leveraging the way thunks are shared between threads. They present a method for leaking one bit of information, along with a technique to amplify the attack. As a solution, they propose a restriction on sharing of thunks between threads, but do not prove noninterference.

Secure-multi-execution [11] guarantees timing-sensitive noninterference by running multiple copies of the program, at the cost of changing the semantics of insecure programs. In order for the secure-multi-execution to provide guarantees against

leakage via memory management it must be necessary to run each copy with a separate collector, effectively enforcing the constraints of Section V. Without such isolation, the shared GC is likely to represent a source of timing channels.

The idea of partitioning heaps based on security levels appears in the work on Relational Hoare Type Theory (RHTT) by Nanevski et al. [27] that provides two different allocation primitives: lalloc for public allocations and alloc for secret ones. Locations obtained by different primitives return disjoint values, which is necessary for defining heap low-equivalence, in many ways similar to our work. However, Nanevski et al. [27] do not consider neither automatic collection nor timing-sensitivity.

The idea of the runtime pc-level is related to the floating label concept in Haskell LIO library [35] that relies on lightweight threads to eliminate internal timing leaks in applications. It is not clear how to combine secure scheduling necessary for concurrent information flow with restrictions on garbage collection that we present here.

Recent years have seen a surge of efforts on verification of garbage collector algorithms and implementations [19, 14]. While these project focus on basic safety properties, they may provide a foundation for design of implementations that would satisfy our security requirements.

X. Conclusion

This paper presents a series of examples that demonstrate feasibility of information leaks via garbage collection. To effectively control such leaks, a tight integration between runtime and the source-level language is needed. We observe that even despite drastic simplifications in the design of the language to simplify aspects such as direct timing attacks, closing leaks via garbage collector requires strong assumptions from the language implementors.

References

[1] J. Agat. Transforming out timing leaks. In *Proceedings of the 27th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '00, pages 40–53, New York, NY, USA, 2000. ACM.

[2] A. Askarov and A. Sabelfeld. Gradual release: Unifying declassification, encryption and key release policies. In *Proc. IEEE Symp. on Security and Privacy*, pages 207–221, May 2007.

[3] A. Askarov, S. Hunt, A. Sabelfeld, and D. Sands. Termination-insensitive noninterference leaks more than just a bit. In *Proceedings of the 13th European Symposium on Research in Computer Security*, pages 333–348, 2008.

[4] A. Askarov, D. Zhang, and A. C. Myers. Predictive black-box mitigation of timing channels. In *ACM Conference on Computer and Communications Security*, pages 297–307, 2010.

[5] M. Assaf and D. A. Naumann. Calculational design of information flow monitors. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 210–224, June 2016.

[6] A. Banerjee and D. A. Naumann. Secure information flow and pointer confinement in a Java-like language. In *Proceedings of the 15th IEEE Computer Security Foundations Workshop*, June 2002.

[7] G. Bollella and J. Gosling. The real-time specification for java. *Computer*, 33(6):47–54, 2000.

[8] P. Buiras and A. Russo. *Lazy Programs Leak Secrets*, pages 116–122. Springer Berlin Heidelberg, Berlin, Heidelberg, 2013.

[9] Z. Deng and G. Smith. Lenient array operations for practical secure information flow. *Proceedings. 17th IEEE Computer Security Foundations Workshop, 2004.*, pages 115–124.

[10] D. E. Denning and P. J. Denning. Certification of programs for secure information flow. *Comm. of the ACM*, 20(7):504–513, July 1977.

[11] D. Devriese and F. Piessens. Noninterference through secure multi-execution. In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 109 –124, may 2010.

[12] C. Dima, C. Enea, and R. Gramatovici. Nondeterministic nointerference and deducible information flow. Technical Report 2006-01, University of Paris 12, LACL, 2006.

[13] D. Frampton, D. F. Bacon, P. Cheng, and D. Grove. *Generational Real-Time Garbage Collection*, pages 101–125. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[14] P. Gammie, A. L. Hosking, and K. Engelhardt. Relaxing safely: verified on-the-fly garbage collection for x86-TSO. In *PLDI 2015: Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 99–109. NICTA, ACM, June 2015.

[15] J. A. Goguen and J. Meseguer. Security policies and security models. In *1982 IEEE Symposium on Security and Privacy, Oakland, CA, USA, April 26-28, 1982*, pages 11–20, 1982.

[16] D. Hedin, A. Birgisson, L. Bello, and A. Sabelfeld. JSFlow: tracking information flow in JavaScript and its APIs. In *SAC '14: Proceedings of the 29th Annual ACM Symposium on Applied Computing*. Chalmers University of Technology, ACM, Mar. 2014.

[17] J. A. Joao, O. Mutlu, and Y. N. Patt. Flexible reference-counting-based hardware acceleration for garbage collection. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, pages 418–428, New York, NY, USA, 2009. ACM. ISBN 978-1-60558-526-0.

[18] R. Jones, A. Hosking, and E. Moss. *The Garbage Collection Handbook: The Art of Automatic Memory Management*. Chapman & Hall/CRC, 1st edition, 2011.

[19] A. McCreight, T. Chevalier, A. Tolmach, A. McCreight, T. Chevalier, and A. Tolmach. *A certified framework for compiling and executing garbage-collected languages*, volume 45. ACM, Sept. 2010.

[20] D. McCullough. Noninterference and the composability of security properties. In *Proc. IEEE Symp. on Security and Privacy*, pages 177–186, May 1988.

[21] S. Moore, A. Askarov, and S. Chong. Precise enforcement of progress-sensitive security. In *Proceedings of the 19th ACM Conference on Computer and Communications Security*, pages 881–893, Oct. 2012.

[22] G. Morrisett, M. Felleisen, and R. Harper. Abstract models of memory management. In *Proceedings of the Seventh International Conference on Functional Programming Languages and Computer Architecture*, FPCA '95, pages 66–77, New York, NY, USA, 1995. ACM. URL http://doi.acm.org/10.1145/224164.224182.

[23] S. Muller and S. Chong. Towards a practical secure concurrent language. In *Proceedings of the 25th Annual ACM SIGPLAN Conference on Object-Oriented Programming Languages, Systems, Languages, and Applications*, pages 57–74, New York, NY, USA, Oct. 2012. ACM Press.

[24] A. C. Myers. *Mostly-Static Decentralized Information Flow Control*. PhD thesis, Massachusetts Institute of Technology, Jan. 1999.

[25] A. C. Myers. JFlow: Practical mostly-static information flow control. In *Proceedings of the 26th ACM Symp. on Principles of Programming Languages (POPL)*, pages 228–241, Jan. 1999. URL http://www.cs.cornell.edu/andru/papers/popl99/popl99.pdf.

[26] A. C. Myers, L. Zheng, S. Zdancewic, S. Chong, and N. Nys-

trom. Jif 3.0: Java information flow. Software release, http://www.cs.cornell.edu/jif, July 2006.

[27] A. Nanevski, A. Banerjee, and D. Garg. Verification of information flow and access control policies with dependent types. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, SP '11, pages 165–179, Washington, DC, USA, 2011. IEEE Computer Society. ISBN 978-0-7695-4402-1.

[28] D. Osvik, A. Shamir, and E. Tromer. Cache attacks and countermeasures: The case of aes. In D. Pointcheval, editor, *Topics in Cryptology – CT-RSA 2006*, volume 3860 of *Lecture Notes in Computer Science*, pages 1–20. Springer Berlin Heidelberg, 2006.

[29] C. S. Pasareanu, Q. S. Phan, and P. Malacaria. Multi-run side-channel analysis using symbolic execution and max-smt. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 387–400, June 2016.

[30] F. Pizlo, L. Ziarek, P. Maj, A. L. Hosking, E. Blanton, and J. Vitek. Schism: fragmentation-tolerant real-time garbage collection. In *ACM Sigplan Notices*, volume 45, pages 146–159. ACM, 2010.

[31] F. Pottier and V. Simonet. Information flow inference for ML. In *Proceedings of the 29th ACM Symp. on Principles of Programming Languages (POPL)*, pages 319–330, 2002.

[32] A. Russo and A. Sabelfeld. Dynamic vs. static flow-sensitive security analysis. In *Proceedings of the IEEE Computer Security Foundations Symposium*, 2010.

[33] A. Sabelfeld and A. C. Myers. Language-based information-flow security. *IEEE J. Selected Areas in Communications*, 21(1):5–19, Jan. 2003.

[34] V. Simonet. The Flow Caml System: documentation and user's manual. Technical Report 0282, Institut National de Recherche en Informatique et en Automatique (INRIA), July 2003.

[35] D. Stefan, A. Russo, J. C. Mitchell, and D. Mazières. Flexible dynamic information flow control in haskell. In *Proceedings of the 4th ACM Symposium on Haskell*, Haskell '11, pages 95–106, New York, NY, USA, 2011. ACM.

[36] G. Tene, B. Iyengar, and M. Wolf. C4: The continuously concurrent compacting collector. In *Proceedings of the International Symposium on Memory Management*, ISMM '11, pages 79–88, New York, NY, USA, 2011. ACM.

[37] D. Ungar. Generation scavenging: A non-disruptive high performance storage reclamation algorithm. In *Proceedings of the First ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, SDE 1, pages 157–167, New York, NY, USA, 1984. ACM.

[38] M. Vaninwegen, C. Gunter, and P. Buneman. The machine-assisted proof of programming language properties abstract the machine-assisted proof of programming language properties. 1996.

[39] D. Volpano, C. Irvine, and G. Smith. A sound type system for secure flow analysis. *J. Comput. Secur.*, 4(2-3):167–187, Jan. 1996.

[40] P. L. Wadler. Analysis of an algorithm for real time garbage collection. *Commun. ACM*, 19(9):491–500, Sept. 1976.

[41] G. Winskel. *The Formal Semantics of Programming Languages: An Introduction*. MIT Press, Cambridge, MA, 1993.

[42] A. Wright and M. Felleisen. A syntactic approach to type soundness. *Inf. Comput.*, 115(1):38–94, Nov. 1994. ISSN 0890-5401. doi: 10.1006/inco.1994.1093. URL http://dx.doi.org/10.1006/inco.1994.1093.

[43] S. Zdancewic and A. C. Myers. Observational determinism for concurrent program security. In *Proceedings of the 16th IEEE Computer Security Foundations Workshop*, pages 29–43, June 2003.

[44] D. Zhang, A. Askarov, and A. C. Myers. Predictive mitigation of timing channels in interactive systems. In *ACM Conference on Computer and Communications Security*, pages 563–574, 2011.

[45] D. Zhang, A. Askarov, and A. C. Myers. Language-based control and mitigation of timing channels. In *Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '12, pages 99–110, New York, NY, USA, 2012. ACM.