Probabilistic Obfuscation through Covert Channels

Jon Stephens Babak Yadegari Christian Collberg Saumya Debray Carlos Scheidegger
Department of Computer Science
The University of Arizona
Tucson, AZ 85721, USA
Email: {stephensj2, babaky, collberg, debray, cscheid}@cs.arizona.edu

***Abstract*—This paper presents a program obfuscation framework that uses covert channels through the program's execution environment to obfuscate information flow through the program. Unlike prior works on obfuscation, the use of covert channels removes visible information flows from the computation of the program and reroutes them through the program's runtime system and/or the operating system. This renders these information flows, and the corresponding control and data dependencies, invisible to program analysis tools such as symbolic execution engines. Additionally, we present the idea of probabilistic obfuscation which uses imperfect covert channels to leak information with some probabilistic guarantees. Experimental evaluation of our approach against state of the art detection and analysis techniques show the engines are not well-equipped to handle these obfuscations, particularly those of the probabilistic variety.**

1. Introduction

This paper describes a novel approach to code obfuscation that uses covert channels, arising from a program's interactions with its execution environment, to conceal information flow in its computation and thereby confuse information flow analyses. The ideas presented can be used for stealthy exfiltration of information in ways that cannot easily be detected using existing techniques.

Obfuscations that utilize covert channels are fundamentally different from other code obfuscations that have been discussed in the literature. Traditional obfuscations come in two flavors: *control flow obfuscation*, which disguises the order in which program statements are executed; and *data obfuscation*, which disguises the values that are manipulated by the computation. The former typically introduce additional information flow paths in order to confuse analyses, while the latter modify the computations in the existing information flows in order to make them harder to untangle. In either case, the original information flows in the program's computation remain, leaving them open to examination by information-flow-based attacks [1]. The covert channel obfuscation techniques we introduce in this paper, by contrast, hide the *presence of data flow*, i.e., data dependencies. The importance of such obfuscation arises from the fact that tracking data dependences is an important component of many security-relevant program analyses (including information flow analysis). By removing information flows from the program's visible computation, our covert-channel-based obfuscations render these flows invisible to program analyses and thereby fundamentally change the attack surface of the obfuscated code.

A second motivation behind this work is the recent emergence of techniques that exploit covert channels to sidestep privacy protections on mobile systems [2], [3], [4], [5], [6]. The research literature typically considers these covert channels used as conceptually distinct and unrelated entities. This paper provides a general framework for reasoning about and understanding covert channels and the information flow obfuscations they enable.

Finally, we introduce the notion of probabilistic obfuscation. It is generally assumed that obfuscating transformations should be semantics preserving. However, there are situations where some semantic slack may be acceptable, e.g., malware writers (who heavily obfuscate their code in order to protect it from analysis) may be perfectly happy if some fraction of the millions of malware instances they distribute fail in the field, as long as they execute correctly "often enough." We refer to obfuscating transformations that are not always completely semantics-preserving as *probabilistic obfuscations*. Such transformations form interesting and novel additions to the obfuscation arsenal. In particular, the construction of *probabilistic obfuscation building-blocks*—such as the covert channel data flow primitives presented in this paper—allow us to construct non-deterministic variants from traditional deterministic obfuscating transformations.

This paper makes the following contributions:

- it describes a semantic framework for understanding information transfer via covert channels;
- it describes a novel approach to obfuscation that removes information flows from a program, rendering them invisible to traditional analysis techniques;
- it describes multiple channels of information flows that can be exploited in this way to impede analysis;
- it shows that current information-flow-based analyses fail to detect the described covert channels; and
- it introduces the idea of *probabilistic obfuscation*

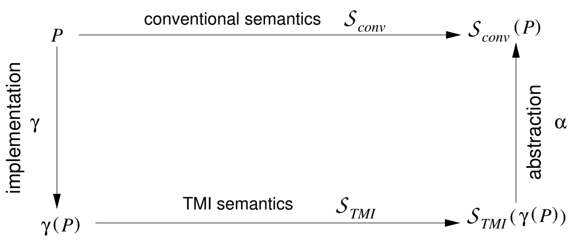

Figure 1. TMI Semantics

and shows that correctness guarantees can be provided even in the presence of imperfect covert channels.

The remainder of the paper is organized as follows. Sec. 2 describes a semantic framework for understanding covert channels. Sec. 3 discusses the attack and defense models assumed in this work. Sec 4 describes the use of covert channels for code obfuscation. Sec. 5 explores the notion of probabilistic obfuscation and correctness guarantees. Sec. 6 presents evaluation results for a prototype implementation. Finally, Sec. 7 discusses related work and Sec. 8 concludes.

2. Semantic Considerations

In order to understand how a program's behavior can be influenced in specific ways by the deliberate use of covert channels, this section gives a brief and informal synopsis of how and where covert channels can arise in the (operational) semantics of a program, which specifies a program's execution behavior in terms of a sequence of state transitions of an abstract machine.

2.1. TMI Semantics

Program execution on a modern computer system involves interactions between many complex components: the program's runtime system, the operating system, the CPU, disk, memory, and various levels of cache. Each component has its own state that affects, and is affected by, the program's execution, and so could plausibly be part of a semantic description of the program's execution. We refer to an operational semantics that gives a detailed picture of a program's execution, encompassing both state changes corresponding to program constructs as well as those corresponding to implementation-level aspects of the program's execution, as *TMI Semantics*.[1]

As the name suggests, TMI semantics can have much more information than necessary. In most cases, such a fine-grained description simply clutters up the semantics and impedes, rather than helps, with understanding the program's behavior. We can get around this problem via an abstraction function α that maps the TMI semantics to the conventional semantics by discarding irrelevant implementation-level detail from the TMI semantics. This is illustrated in Figure 1. Note that, depending on the amount of detail captured, a given program can have many different "conventional semantics" and many different TMI semantics; correspondingly, there is a different abstraction function α for each $\mathcal{S}_{TMI}$ and $\mathcal{S}_{conv}()$. The arrow labeled 'implementation' in Figure 1 should not be understood as mapping each source program to a unique implementation. A given source program can have many different implementations, e.g., corresponding to different compilers or compiler optimizations. Indeed, correctness of the compiler requires that, for every program P and any pair of implementations γ_1, γ_2 of P, it must be the case that $\alpha(\mathcal{S}_{TMI}(\gamma_1(P))) = \alpha(\mathcal{S}_{TMI}(\gamma_2(P)))$.

1. "TMI" stands for "Too Much Information."

2.2. Visible vs. Invisible State

In general, states in the TMI semantics consist of many different components. For a state in the execution of a given program, for example, these may include: the program counter; values for (memory locations and registers corresponding to) its variables; the runtime clock; and information about the internal state of the runtime system, e.g., the garbage collector, heap memory allocator, and JIT compiler. The abstraction function α shown in Figure 1 discards information about some of these components; components of the TMI semantics state that are discarded by α are invisible in the conventional semantics. We refer to such components as being in the "invisible state." More formally, a component I of a TMI semantics $\mathcal{S}_{TMI}$ is *invisible* under an abstraction function α if there exist two TMI states s_1 and s_2 that differ on the value of the component I but where $\alpha(s_1) = \alpha(s_2)$, i.e., the fact that s_1 and s_2 differ on the value of component I is not visible once we apply the abstraction function α. For example, if an abstraction function discards information about the cache behavior of a program, then two TMI states that differ only on whether or not a particular memory location is in the level-1 cache will not be distinguishable under α; in this case, therefore, the component of the TMI semantics corresponding to the cache will be part of the invisible state. As another example, consider two TMI states in an interpreted system that differ only on whether or not a particular function has been JIT-compiled. If α discards information about the execution speeds of functions, the "JITted-ness" of functions will be part of the invisible state. An abstraction function α thus induces a partitioning of the components of each TMI state s into a "visible" part, which is reflected in $\alpha(s)$, and an "invisible part" that is not reflected in $\alpha(s)$.

2.3. Code Obfuscation via Covert Channels

A code obfuscation tool takes a program P as input and transforms it into a program P', semantically identical to P. The goal is for P' to be much less amenable to analysis than P, while minimizing the computational overhead incurred. There are many aspects of P that we may want to obscure, such as control flow, abstraction layers, embedded secrets such as cryptographic keys, etc. Traditional

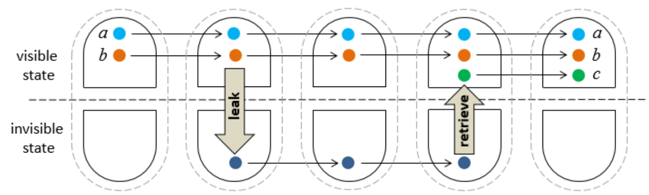

Figure 2. Information flow through visible and invisible states. The value a represents "normal" flow and occurs entirely through visible states; the value b flows through visible states, but is also injected into the invisible state and subsequently retrieved into c. The flow from b to c occurs through a covert channel.

obfuscating transformations include virtualization, flattening [7], branch functions [8], white-box cryptography [9], and data encoding [10]. On the theory side, there exist both impossibility results [11] and recent results showing that cryptographically secure obfuscation is possible under some models [12], albeit with unacceptably high levels of computational overhead.

In a conventional view of program execution, values that are computed and propagated by the computation flow through components of the visible state. The key idea behind using covert channels for code obfuscation is the insight that the flow of values through a computation need not always go through the visible part of a state, but can sometimes occur through the invisible part.

In order to realize this functionality, we have to inject information from the visible state into some component of the invisible state, and subsequently recover information from that component of the invisible state back into the visible state. To this end, we propose two primitives: $\mathbf{leak}_k$, which injects information about a visible-state value a into the invisible-state component I_k by perturbing the value of I_k in a way that captures some aspect of the value a; and **retrieve**, which returns to the visible state a value retrieved from the invisible-state component of the program's state. This is illustrated in Figure 2, where the **leak** primitive injects the value b from the visible state into the invisible state, and the **retrieve** primitive later retrieves this value back into the visible state (possibly into a different variable c). For example, if the component I_k is the byte code of the program in an interpreter, then $\mathbf{leak}_k$ may use a bit in the value of a visible-state variable x to cause some function in the program to become JIT-compiled, and **retrieve** may use the execution time of the function to determine whether it has been JIT-compiled and thereby reconstruct the value of the corresponding bit of the variable x.

In order to be useful, **retrieve** should return the information leaked by **leak**: namely, for all values a, $\mathbf{retrieve}_k(\mathbf{leak}_k(a)) = a$; or, equivalently, they should compose to the identity function:

$$\mathbf{retrieve}_k \circ \mathbf{leak}_k \equiv \mathbf{id}.$$

If this condition is satisfied, the invisible-state component I_k forms a usable covert channel. As discussed in Section 5, it may be possible to relax this requirement so that it holds probabilistically, in which case we get a probabilistic covert channel.

Later sections of this paper give several examples of covert channels realized by using different components of the invisible program state together with the corresponding **leak** and **retrieve** functions.

3. Attack and Defense

For the obfuscation to be successful, some adversary must not be able to identify the software that leaks information using our obfuscation, given some universe of programs, with a high degree of confidence. This adversary has unlimited resources and complete control over the system, allowing them to observe every action taken by the software (including in the kernel).

3.1. Attack Model

Covert channels can be detected by: either (i) identifying the covert channel primitives, or (ii) identifying perturbations from normal behavior. Identifying the covert channel primitives requires discovering at least one of the following components: (i) a program construct A that affects an invisible-state component I_k in a way that depends on a visible-state value (**leak**); (ii) a construct B that computes a value that is dependent on the same invisible-state component I_k (**retrieve**); and (iii) reachability of B from A via the program's control flow. Identifying perturbations, on the other hand, requires monitoring for statistical anomalies.

3.1.1. Identifying Primitives. Both the leak and retrieve primitives can be detected using static program analysis. The adversary can look for one or both of the primitives directly in the binary or source code. For example, to detect a timing channel through the file cache, an adversary may look for timed file writes.

Another available avenue to detection is for the adversary to observe the flow of information from the leak primitive to the retrieve primitive using dynamic information flow techniques such as taint analysis. Static information flow does not make sense in this context since static analyses cannot reason about actions taken outside of the source they are operating on. Since the obfuscation leaks information through the runtime system, the flow will not be visible in the source and therefore requires all covert channels in the runtime system to be simulated. A dynamic analysis, on the other hand, can be performed at a much lower level and therefore requires much less simulation.

3.1.2. Identifying Perturbations. An individual channel's statistical behavior can be monitored for abnormalities (as is common for network covert channels [13]). This approach creates a model for the typical behavior of the target leak primitive and compares it to the behavior it observes in other software. Anything that deviates too far from the model will then be flagged as potentially obfuscated.

3.2. Defense Model

While there are several detection methods available to the adversary, there are defenses that augment the obfuscation to make it more difficult to detect.

There are several problems with using static program analysis to detect covert channels. First, since it simply looks through an executable for the leak and retrieve primitives, there is no way of discovering new covert channels using this approach. The adversary must therefore know all of the exploitable covert channels available on their system to consistently detect our obfuscation as it is not tied to a single channel. Second, the attack will only work if the primitives used for covert channels are unstealthy, i.e., not commonly used in typical software. Our framework, however, does not rely on a single primitive to leak or retrieve values and is easily extensible so that more can be added in the future. This flexibility allows us to swap infrequently used primitives for more frequently used ones that are harder to detect. For example, it may be trivial to detect timed file reads, but our framework can instead make use of implicit timing using threads (see Section 4.4), resulting in behavior that is common in typical software.

Information flow techniques can be applied to detect covert channels, but since the information flows under consideration have been moved out of the program by the obfuscations and into the runtime system and/or operating system, the amount of code that has to be considered has to encompass all of this code as well. Such analyses therefore require an enormous implementation effort due to the amount of state that must be simulated. The adversary must be capable of observing all state affected by the actions of a program so that they can detect all information flows in the system. As a result, they must simulate state at the hardware and network level which, while theoretically possible, requires substantial effort. Therefore, typically the lower the covert channel communicates in the runtime system, the more difficult it will be to detect since more elements of the system will have to be observed or simulated.

Statistical analyses assume that the adversary knows and can monitor all covert channels in the system and that the usage of the covert channel will result in a meaningful deviation from the model derived by the adversary; however, as discussed by Crespi *et al.* [14], it is possible to leak information without violating the adversary's model. To do so, the obfuscation can construct its own statistical model of the channel and modify its behavior based on those statistics to evade detection. As long as the model derived by the obfuscator is at least as accurate as the adversary's model, detection can be avoided.

4. Obfuscating Data Flow

As mentioned earlier, traditional obfuscations retain the information flows in the program, but augment and/or modify them in order to make them harder to analyze. This section describes a family of obfuscating transformations that takes a fundamentally different approach: it *conceals* data flow by moving them out of the computation and into the program's runtime system or the operating system. While some of our transformations incorporate ideas previously discussed in the covert channel literature, they are employed in a radically different way: rather than using covert information flows as an attack to exfiltrate information from a computing system, we *protect* a computing system by using covert flows to hide the presence of data flow from analysis.

The transformations are designed such that a programmers can trade off between complexity, diversity, stealth, level of semantic preservation, and performance. The transformations have been incorporated into a publicly available C source-to-source obfuscation tool that is capable of transforming real programs written in the `gcc` dialect of C.[2]. The obfuscator supports a large collection of traditional transformations [15] in addition to the ones presented here: virtualization, dynamic obfuscation (self-modifying code), branch functions, control-flow flattening, etc. These transformations can be freely mixed-and-matched, allowing, for example, code obfuscated with the transformations proposed in this paper to be further transformed by adding a layer of virtualization, then again transformed by replacing direct branches by calls to branch functions, etc.

In the following, we will first describe *deterministic primitives* which are well-known techniques from the literature to obscure data flow. Next, we will present a set of novel *non-deterministic primitives* which have been inspired by the idea of timing-based covert channels. We next present a set of *combiners* that allow us to connect deterministic and non-deterministic primitives in ways that result in obfuscated data flow of arbitrary complexity and level of semantic precision. We next present ways to compute time in our timing-based covert channels without explicitly using the timing facilities provided by the operating system, since such code will be unstealthy in many programs. We conclude the section with a discussion of practical concerns.

4.1. Deterministic Primitives

Primitive 1 Increment

```
void P() {                    void P'() {
   int a = b;         ⇒          int i,a = 0;
}                                for(i=0;i<b;i++)
                                    a++;
                              }
```

Simple deterministic data flow-obfuscating primitives have been described previously in the literature [16], [17], [18]. Primitive 1 shows a trivial example where `a` is incremented up to the value of `b`. While there is no direct data-flow dependence on `a` and `b`, there *is* a control-flow dependence. This is known as *implicit flow* and analyses exist to handle the implicit flow [19], [20], [21].

2. The tool itself and all the test cases referred to in this paper can be downloaded from https://github.com/triple-blind/submission-1 The executable binary of the tool will be publicly available from our website; its source code will be available to researchers on request.

Primitive 2 Signals

```
unsigned int value;
int bitNo;
void handler(int sig){
  value |= 1 << bitNo;
}
void P'() {
   value=0;
   signal(31, handler);
   for(i in [0...(bits in b)-1]) {
      if ((i^th bit of b)==1) {
        bitNo=i;
        raise(31);
      }
   }
   signal(31, 1);
   a = value;
}
```

A different technique uses signals to, one bit at a time, copy the value of `b` into `a`. This is shown in Primitive 2. Again, techniques have been developed to analyze such codes [22].

It should be noted that for each of these techniques many variants are possible, and more variants will add to the diversity of the obfuscated code. Our current implementation includes 7 deterministic primitives but many more are possible, and the architecture of the obfuscator is such that it is easy to plug in new variants.

Primitive 3 Non-deterministic primitive

```
1  void P'() {
2     value=0;
3     for(i in [0...(bits in b)-1]) {
4        timeT start = time();
5        if ((i^th bit of b)==1)
6           slow process(param);
7        else
8           fast process(param);
9        timeT time = time()-start;
10       if (time > threshold)
11          value |= 1 << i;
12    }
13    a = value;
14 }
```

4.2. Non-Deterministic Primitives

We next describe a method to hide the assignment `a = b` that neither displays direct data flow nor implicit control flow. The idea is to encode the value of a bit to be copied in the time it takes to execute a particular process. Conceptually, `a = b` is transformed into the code in Primitive 3. Lines 5-8 correspond to the **leak** function of Section 2, and lines 4, 9-11 to the **retrieve** function.

This idea was inspired by attacks on the side channels found in the implementation of many cryptographic algorithms. In such attacks, bits of a secret are extracted by measuring artifacts of the execution of the algorithm, such as time, energy, or electromagnetic radiation [23], [24]. Our system is similar in that it moves information (bits of a word to be copied) using execution artifacts, but different in that the measurement of the artifact is *internal* to the program, not external. In the following we will restrict our measurements to time, since this is readily available from inside a program, but other channels are certainly possible, and the principles remain the same.

While it would be trivial to generate two processes where one is slower than the other—two loops with different bounds would suffice—this would not sufficiently raise the bar for the adversary. Instead, we will seek processes that exploit aspects of the underlying hardware, operating system, and runtime system. Again, this is inspired by proposed side channel attacks, such as those that make use of processor caches [25]. Our ultimate goal is to force the adversary to encode, in their analysis tools, not only the semantics of the instruction set and system calls, but also *extra-semantic* characteristics of the entire platform, such as the behavior of instruction caches, file caches, garbage collectors, jit compilers, etc.

Primitive 4 Data cache

```
posix_memalign(&buf1,pagesize,64);
posix_memalign(&buf2,pagesize,64);
slow process(param):
   for(i=0;i<param;i++){
      asm ("mfence\n"
      "clflush (%0)\n"::"r"(buf1));
      sum += *((long *)buf1);
      *((long *)buf1) = sum;
   }
fast process(param):
   for(i=0;i<param;i++){
      asm ("mfence\n"
      "clflush (%0)\n"::"r"(buf1));
      sum += *((long *)buf2);
      *((long *)buf2) = sum;
   }
```

4.2.1. Data cache channel. Our first channel will make use of characteristics of processor data caches. Conceptually, depending on the value to be transmitted, the **leak** function loads the content of an address into the processor cache or flushes the cache line at that address. To recover the value, the **retrieve** function measures the time taken to load the data at that address.

Our tool generates the code in Primitive 4. Note that the only difference between the fast and the slow processes is how they treat the two buffers, `buf1` and `buf2`. The slow process first flushes `buf1` and then reads from it, forcing the processor to reload the corresponding cache line. The fast process, however, flushes `buf1` and then reads from `buf2` which is (likely to be) mapped to a different cache line and thus, after the first read, likely to be cached.

All our non-deterministic primitives have a tunable parameter (*param* in Primitive 4). These need to be adjusted such that the difference in timing between the slow and the fast processes is significant enough that it can be effectively measured given the resolution of the clocks used, and also consistently producing the correct result, given normal fluctuations on the platform. We will discuss training of the parameters later in this section.

Primitive 5 File cache

```
process(param,nocache):
   posix_memalign(&buf,pagesize,
                  pagesize);
   fd=open("/tmp/file.txt", writing);
   fcntl(fd, F_NOCACHE, nocache);
   for(i=0; i<param; i++)
     write(fd,buf,pagesize);
   close(fd);

   fd=open("/tmp/file.txt", reading);
   fcntl(fd, F_NOCACHE, nocache);
   start = time();
   for(i=0; i<param; i++)
     read(fd,buf,pagesize);
   time = time()-start;
   close(fd);
   unlink("/tmp/file.txt");

slow process(param):
   process(param,1);
fast process(param):
   process(param,0);
```

4.2.2. File cache channel. In order for an analysis tool to process the code in Primitive 4, it needs at least a rudimentary understanding of the runtime behavior of CPU caches. We would like to force the analysis tool to have an understanding not just of the hardware, but of the behavior of *every* level of the complete platform, including the operating system. Our second non-deterministic primitive is also based on caching, but makes use of file caches rather than instruction caches. Here, the **leak** function transmits a value by conditionally loading a file into the file cache, and the **retrieve** function recovers that value by the time it takes to read the file. Our obfuscator generates the code in Primitive 5.[3]

4.2.3. Jitting channel. Many programs today are interpreted and, in order to reduce the performance overhead of interpretation, a *just-in-time* (JIT) translator (included with the run-time system) compiles the interpreted bytecode to machine code on the fly. Typically, in order to avoid the overhead of interpretation, the run-time system will interpret a function the first few times it is called and, only when it has decided the function is indeed a hotspot, will it invoke the JIT translator.

Our obfuscator includes a runtime JIT translator which can be used by itself as an advanced *packer* transformation that only produces machine code for a function when it is called. The JITter also forms the basis for dynamic obfuscating transformations that generate self-modifying code. We make use of this JITter to construct a **leak** function that transmits a value by conditionally JIT-compiling a particular procedure and a **retrieve** function that recovers the value by timing a call to the procedure.

The resulting code is shown in Primitive 6. Here, the slow process measures the time of *both* the JITter translating `foo` to machine code *and* the call to the jitted function `foo` itself. The fast process, on the other hand, only measures, the time of the JITted function. In this example the JITting always happens the first time a function gets called but this can be varied to make the process less predictable to an analysis tool.

Primitive 6 Jitting

```
int freq=0;
void foo(input,output) {
   static void (*foop)(...,...);
   if (freq==0) {
      foop = JIT(bytecodes);
      freq++;
   }
   (*foop)(input,output);
}
slow process(param):
   freq=0;
   start=time();
   foo(...,...);
   time=time()-start;
fast process(param):
   freq=0;
   foo(...,...);
   start=time();
   foo(...,...);
   time=time()-start;
```

Primitive 7 Garbage collection

```
process(size):
   GC_gcollect();
   buildLinkedList(size);
   timeT start = time();
   GC_gcollect();
   timeT time = time() - start;
slow process():
   process(large number);
fast process():
   process(small number);
```

4.2.4. Garbage collection channel. Many modern languages include a garbage collector as part of the runtime system. This gives us yet another subsystem on which to build a timing channel. Many possibilities avail themselves, especially if the particulars of the garbage collection algorithm are known. For example, a copying collector is expected to be faster when the heap consists mostly of garbage than when every object is reachable from the roots. Therefore, we can create a **leak** function that transmits a value by varying the reachability of objects on the heap and

3. The code shown is for MacOS/Darwin. The Linux interface is different, passing the *nocache* flag to the `open` system call, rather than to `fcntl`. Our obfuscator supports both Linux and MacOS.

a **retrieve** function that collects that value by timing the garbage collector.

In the example in Primitive 7 we are using the Boehm mark-and-sweep collector [26][4]. This code first performs a collection to clear the heap of any existing garbage. Next, a linked list is created, a long one for the slow process and a shorter one for the fast process. Finally, a second garbage collection is performed and timed. With a mark-and-sweep collector a garbage collection of a heap containing of a very long chain of reachable objects is expected to be slower than a collection with fewer reachable objects.

4.3. Flow Combiners

There are several issues with the flow primitives we have described so far. While there are undoubtedly many possible techniques yet to be discovered to hide data flow, deterministic as well as non-deterministic, the number of such techniques is likely to be finite. This is a problem since it puts a practical limit on the level of diversity that an obfuscation tool can achieve. Furthermore, our timing-based primitives by their very nature will sometimes fail, i.e. the process meant to be fast will, occasionally, be confused for a slow process. This will result in an assignment `a = b` giving `a` the wrong value and likely causing program failure. However, we would like the failure mode to be under the control of the programmer who is in the best position to make the appropriate trade-offs between performance, diversity, and correctness.

To these ends, we introduce the concept of *flow combiners*. These are operators which can compose the primitives described above to achieve desired levels of diversity and correctness. Combiners are recursive, meaning they can be applied *ad infinitum*. Our system currently supports 5 combiners:

combiner ::=
  | *primitive*
  | *compose*(list of *combiner*)
  | *select*(list of *combiner*)
  | *majority*(list of *combiner*)
  | *repeat*(*combiner*, int)
  | *until*(*combiner*, int, int)

We describe the semantics of combiners by example. In the following, let d_i be deterministic and n_i non-deterministic primitives, and c_i any combiner. As before, let us transform the assignment `a = b`. The combiner $\mathit{compose}(c_0, c_1, \ldots)$ chains together several combiners, i.e. the output of combiner c_i becomes the input to combiner c_{i+1}, meaning `a = b` is transformed into `a =` $c_0(c_1(\ldots(c_i(\mathtt{b}))))$. The $\mathit{select}(c_0, c_1, \ldots, c_{n-1})$ combiner will choose one of its constituent combiners at random:

```
switch (random symbolic
expr % n) {
   case 0: a = c0(b)
   case 1: a = c1(b)
   ...
}
```

In our implementation the combiner is chosen pseudo-randomly, and dependent on input, making the variable `a` a symbolic variable. The combiner $\mathit{majority}(c_0, c_1, \ldots)(\mathtt{b})$ will compute $c_0(\mathtt{b}), c_1(\mathtt{b}), \ldots$ and choose the most frequently occurring value. $\mathit{repeat}(c_0, n)$ is equivalent to $\mathit{majority}(\overbrace{c_0, c_0, \ldots}^{n})$. The combiner $\mathit{until}(c_0, m, n)$, finally, continuously repeats $c_0(\mathrm{b})$ m times until there is at least n agreements on the resulting value.

Flow combiners allow us to generate arbitrarily complex flow expressions. For example, `a = b` can be turned into

$$\mathtt{a} = \mathit{majority}(\mathit{compose}(d_0, n_0), d_1, \mathit{repeat}(n_1, 5))(\mathtt{b}),$$

They also allow us to increase our confidence in non-deterministic primitives by combining them using majority logic:

$$\mathtt{a} = \mathit{majority}(n_0, n_1, n_2, n_3, n_4)(\mathtt{b})$$

Furthermore, we can create deterministic flow expressions from non-deterministic primitives, by combining them with deterministic ones. For example, the flow expression

$$\mathtt{a} = \mathit{majority}(n_0, n_1, d_0, d_1, d_2)(\mathtt{b})$$

will always compute the correct value for `a`; should one or both of n_0 and n_1 fail they will still be outvoted by d_0, d_1, d_2. Finally, *select* and *repeat* allow us to balance correctness and performance overhead:

$$\mathtt{a} = \mathit{select}(d_0, d_1, d_2, \ldots, \mathit{repeat}(n_0, 7))(\mathtt{b})$$

Here, we will mostly execute deterministic primitives (which tend to be fast) mixed with the occasional (slower) non-deterministic primitive.

4.4. Stealthy Timing Without Timing

One issue with the timing-based primitives we have seen is that the act of a program timing itself may be unstealthy in many programs. While our implementation supports several timing primitives (such as the X86 `RDTSC` *read timestamp counter* instruction and the `gettimeofday` system call), neither is likely to be frequently occurring in many programs. This could leave our obfuscated code open to static program analysis attacks as discussed in Section 3.1.

We therefore introduce the ability to replace explicit timing with *implicit timing* using threads. The code in Primitive 8 illustrates the basic idea. Here, we spawn a slow and a fast thread, these threads write (with a deliberate race condition) on the variable `result`, and, finally, a barrier waits for both threads to finish. At the end, the slow thread is likely to have finished last thereby assigning the correct bit to `result`.

4. We have tested the Boehm library to ensure that it forms a feasible garbage collection channel, but we have not yet integrated it into our obfuscation tool.

Primitive 8 Thread-based timing

```
int result;
void threadZero(void (*work)()) {
   work();
   result = 0;
}
void threadOne(void (*work)()) {
   work();
   result = 1;
}
void P'() {
   a=0;
   for(i in [0...(bits in b)-1]) {
      zeroWork = slow process;
      oneWork = fast process;
      if(i^th bit of b) {
          zeroWork = fast process;
          oneWork = slow process;
      }
      s=thread_create(threadZero, zeroWork);
      f=thread_create(threadOne, oneWork);
      thread_join(s);
      thread_join(f);
      if (result)
         a |= 1 << i;
   }
}
```

While the code in Primitive 8 should fit in many threaded programs (high performance codes often use `create` and `join` in this way), depending on the threading behavior of the input program, this design too may be unstealthy! Fortunately, there are many variants of the basic idea that can be matched to the threading behavior of a particular program. For example, we have a variant that uses a *thread-pool* (obviating the need for multiple conspicuous `creates`) and a variant that spawns only one thread instead of two. Even single-threaded programs can be accommodated by introducing bogus decoy threads. Finally, suspicious race conditions on the `result` variable could be detected [27], but such potentially unstealthy behavior can be avoided by introducing bogus locks.

4.5. Training Primitives

Before we can use a non-deterministic primitive n we have to *train* it. This means determining two values, *param* and *threshold*, such that the accuracy of n is maximized and the performance overhead is minimized.

Figure 3 shows a case where we have trained Primitive 5 on a modern laptop. We executed 500 slow and 500 fast samples for each parameter value, here represented by circles and triangles respectively. The y-axis shows the value of the parameter (file size, in this case) and the x-axis the number of CPU ticks, as measured by the x86 instruction `RDTSC`. The vertical bars are the thresholds, here computed as the midpoint between the medians of the slow and fast samples. Next to each sample is shown two numbers s/f, the number of slow and fast failures, where fast failures fall to the right of the threshold and slow failures fall to the left.

In our current prototype implementation training proceeds by examining increasingly larger parameter values, until one is found for which there is a suitable "gap" between fast and slow samples. Such a gap will allow for some slack in timing measurements at runtime. In Figure 3, for example, parameter values less than 16 seem to overlap too much, whereas *param=16* or *param=32* display suitable gaps.

4.5.1. Offline vs. On-Demand Training. A question that arises is *when* to train for suitable parameter values. There are three possibilities: *offline* training determines parameter values at obfuscation time, before the program is distributed to its users (and potential adversaries); *startup* training runs when the program is first executed, but before user code starts running; and *on-demand* training mixes the training with the execution of user code. Offline training has the advantage that it has no impact on performance, but suffers from the problem that it cannot know the machine characteristics of all the platforms on which it may potentially run. Startup and on-demand training have the advantage that they run on the actual platform but, as a result, they will suffer performance overhead, either when the program starts up, or during execution. On-demand training has the further advantage that it can adjust parameter values and thresholds as the program is running, potentially taking into account changing runtime characteristics of the program and the environment on which it is running. Our current implementation supports offline and startup, but not on-demand, training.

5. Probabilistic Obfuscation

In order for probabilistic obfuscation to become a viable technique, each non-deterministic transformation must be accompanied by a *correctness guarantee*, i.e. a bound on its failure rate.

In the remainder of this section we will explore such guarantees for the primitives in Section 4. The training routines in our current proof of concept implementation choose parameters and thresholds heuristically, simply looking for the smallest parameter with a "reasonable" gap between fast and slow measurements. While this works well in practice, we would prefer to be able to make statements such as, "after transformation with non-deterministic primitive n, the copy `a = b` will fail no more than once in a million." This would also let the programmer compare different primitives, allowing them to pick one with the correctness guarantees and performance characteristics appropriate for their situation.

5.1. Correctness Guarantees

Although there will always be a small probability that the an assignment $a = b$ transformed with one of the primitives in Section 4 will go wrong, we can choose the desired reliability of the process through two parameters: the target confidence level (which we will call conf), and the target expected error rate (which we will call r). Our

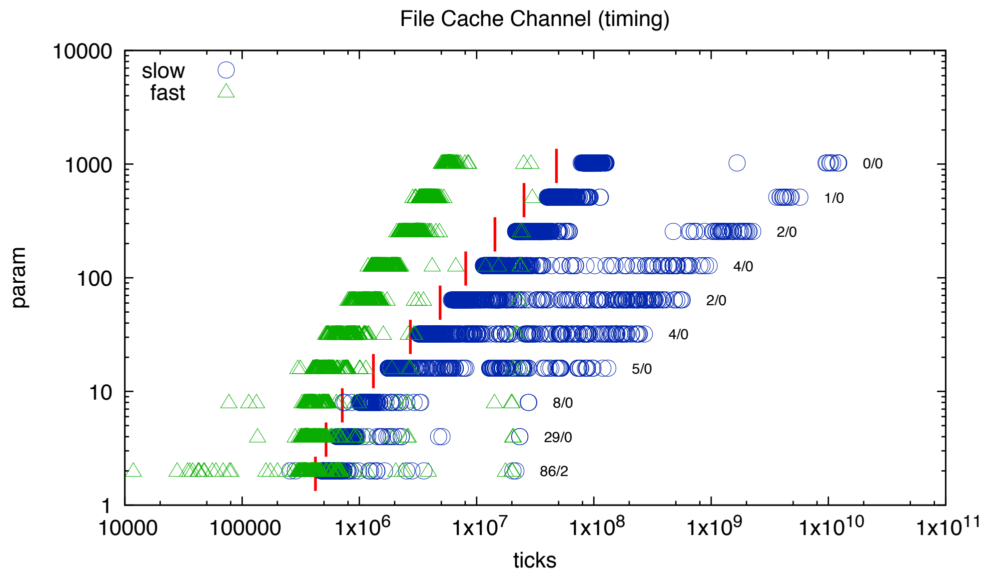

Figure 3. Training results for Primitive 5 in Section 4.2.2. Timings were collected on a laptop with a 2.9GHz Intel Core i7 with 16GB of main memory and 2TB of SSD disk. Both axes are base-10 log scales.

```
bit b = ...;
int n = ⌈2 · log(r)/log(t_i)⌉;
timeT[n] times;
for(i=0;i<n;i++) {
    timeT start = time();
    if (b == 1)
        slow process(param);
    else
        fast process(param);
    times[i] = time()-start;
}
timeT m = median(times);
bit a = m > threshold;
```

Figure 4. Copying a single bit with correctness guarantees.

claim is that the procedure described below will, at the given confidence level, set the variable a incorrectly at a rate that is, at most, the expected error rate. For example, at $\mathrm{conf} = 99\%, r = 10^{-6}$, we can expect that at most once every 100 times the training procedure runs the code will generate errors at an observed rate of more than one in a million.

During training we determine three values, T_i, t_i, and *threshold*, for each non-deterministic primitive n_i. t_i is the estimate of the upper bound of the confidence interval of a Bernoulli random variable at the confidence level conf (using the Wilson score rule [28]). This is determined by tallying the total number of successful and failed transmissions during training. T_i is the expected runtime for one execution of n_i, which we estimate by computing the mean runtime over the training data.

To copy a single bit b to a with the expected error rate r we need to modify our copy procedure so that the bit is sent *multiple times* (see Figure 4).

As a concrete example, consider a situation where during training we have run 500 tests and determined that 50 of them fall on the "wrong side" of the threshold. I.e., in 10% of the tests a slow value was measured as fast or a fast was measured as slow. In this case, we have 450 successes and 50 failures. By using Wilson's score rule, we find that at our chosen confidence level $\mathrm{conf} = 0.99$ and $t_i \approx 0.14$. Suppose that $r = 10^{-6}$, i.e. we are looking for a one in a million error rate. Then, $\log(r) = -6$ and $\log(t_i) \approx -0.857$, so $n = \lceil 2 \cdot \log(r)/\log(t_i) \rceil = 15$. Thus, we need to send each bit of a word 15 times to get a one in a million error rate. For a one in a *billion* error rate, we get a minor increase, namely $n = 21$. To see why the algorithm works, notice that every transmission has at most a t_i chance (at a given confidence level) of being wrong. In order for the decision using the median value to be incorrect, then, more than half of the transmissions need to be wrong. We want to bound this probability, and so we solve for n in $t_i^{2n+1} \le r$ (we take the product of the probabilities since runs of t_i are independent of each other). A small simplification of the solution then gives the expression for n above.

The claim above works for any technique. But how do we decide between techniques? We define a cost in runtime for each reliable transmission, and pick the one that minimizes it. That cost is simply $n \cdot T_i$, or $2 \cdot \log(r) \cdot T_i/log(t_i)$. Since r and t_i are both less than 1, it's easier to work with the negative of their logs, so we get $2 \cdot (-\log(r)) \cdot (T_i/-log(t_i))$. The quantity $T_i/(-log(t_i))$ characterizes each primitive n_i: our goal is to minimize T_i, and maximize $-\log(t)$. It also characterizes the trade-off. Note that this trade-off will make us want to pick primitives for which the individual error rate is relatively high, since they are likely to be much faster. See Figure 3 for typical runtime and error rates.

6. Evaluation

We evaluated a prototype implementation of our framework to assess its effectiveness and performance. We considered two kinds of attacks on our obfuscations. First, since our obfuscation uses covert channels to conceal information flows, we analyzed samples obfuscated using our framework with modern symbolic analysis engines, which are widely used and represent the state of the art in information-flow-based program analysis techniques. Second, in order to evaluate the stealthiness of our approach, we used analyses proposed in the literature aimed specifically at detecting covert channels. There is a very wide variety of techniques that have been proposed in the literature, and as a matter of practicality we implemented and evaluated a couple of such techniques that are very general and broadly applicable against a broad spectrum of covert channels.

6.1. Symbolic Analysis

We evaluated our obfuscation on four state-of-the-art symbolic analysis engines to determine how much functionality is built into typical symbolic analysis tools and frameworks. Two of these, S2E [29] and Fuzzball [30], are complete analysis tools targeted at test case generation, so they symbolically execute a program in order to maximize code coverage. The other two engines, ANGR [31] and Triton [32], are binary analysis frameworks with built-in symbolic execution engines. They provide the user with more flexibility by allowing analyses to be customized. As a result, users can perform more than just symbolic execution with these frameworks, and can customize the way in which the program is symbolically explored. For our evaluation, we wanted to see what could be detected by the two frameworks by default, and therefore what the user would have to add. We therefore configured ANGR and Triton to act like a symbolic analysis tool so that they would provide us with concrete values for the symbolic variables that result in unique paths being taken. Among the tools we experimented with, only S2E is able to trace into kernel code and continue the analysis within the operating system.[5]

6.1.1. Evaluation Process. From the simple input program in Figure 5 our obfuscation tool generated 18 obfuscated samples for the symbolic analysis engines to analyze. The program takes an input, stores it into the variable *a*, and assigns to *b*, thereby creating a direct data dependency. The value in *b* is then compared to the 32-bit constant 0x55787855 (ASCII `"UxxU"`, a number with an equal number of 0 and 1 bits) and, based on the result of the comparison, prints "SUCCESS" or "FAILURE". Therefore,

5. We configured Triton with Intel PIN which does not provide kernel code access to the analysis.

```
int a;
int main(int argc, char *argv[]) {
    a = *(unsigned int *) argv[1];
    int b = a;
    if (b == 0x55787855)
        printf("SUCCESS");
    else
        printf("FAILURE");
    return 0;
}
```

Figure 5. Original input program that was obfuscated for the effectiveness evaluation.

if we mark *a* as symbolic, the engines should be able to find two paths in the executable, one leading to "SUCCESS" when the value of the symbolic variable is 0x55787855, and one leading to " FAILURE" otherwise. As seen in Table 1, all symbolic analysis engines achieve exactly this result when run on the program in Figure 5. Since we know that each of the engines achieve the correct result for our input, we know that if they cannot detect the same paths in our obfuscated programs then the obfuscation must be successful.

To test our framework, we generated the samples described in Table 1 which obfuscate the direct data assignment from *a* to *b* of the program in Figure 5. The samples are organized into 3 classes: deterministic, non-deterministic, and combined. The deterministic samples perform transformations on the data flows that deterministically yield the input value, allowing us to establish if the engines can follow simple obfuscated data flows. The non-deterministic channels then check if any of the engines are capable of tracing information flow through time, either implicitly or explicitly. The final class uses flow combiners to compose the various channels so that we can evaluate if combining flows makes the obfuscation more or less detectable.

To ensure the samples could be run by all of the symbolic analysis engines, some engine-specific additions needed to be made. No modification was necessary for ANGR and Triton since they allow the inputs to a program to be symbolic. Unlike ANGR and Triton, Fuzzball introduces symbolic variables by allowing a user to specify a region in memory to be symbolic. We therefore modified this program to print out the address of the global variable *a*. Additionally, we took out the assignment to *a* from the program's input value since that would overwrite the symbolic variable. Finally, Fuzzball would only analyze 32-bit binaries. This constraint was not an issue for most of the samples, but we were unable to build a 32-bit version of the JIT compiler obfuscation used in sample 13, so it was not evaluated on Fuzzball. S2E introduces symbolic variables and controls path exploration using annotations to the source code. These annotations were added to each sample so that S2E marked *a* as symbolic after the assignment from input and explored the remainder of the program.

Table 1. COLUMNS 3-6 SHOW THE EFFECTIVENESS OF RUNNING SYMBOLIC ANALYSES ON A SET OF BENCHMARK OBFUSCATIONS. A ✓ INDICATES THAT THE SYMBOLIC ANALYSIS ENGINE SUCCESSFULLY DISCOVERED ALL PATHS, WHILE A ✗ INDICATES THAT THE ENGINE WAS NOT SUCCESSFUL. THE SUBSCRIPTS PROVIDE ADDITIONAL INFORMATION: $✗_n$=n FALSE POSITIVES; $✗_{\mathrm{to}}$=TIME OUT; $✗_{\mathrm{k}}$=KILLED BY OS; $✗_{\mathrm{pc}}$=PYTHON INTERPRETER CRASH; $✗_{\mathrm{mmap}}$=UNSUPPORTED MMAP OPERATION; $✗_{\mathrm{uv}}$=UNDECLARED VARIABLE ERROR; $✗_{\mathrm{sys}}$=UNSUPPORTED SYSTEM CALL; $✓_n$= n PATHS FOUND, 2 OF WHICH ARE CORRECT . COLUMN 7 SHOWS THE WALL CLOCK TIMES (IN SECONDS, AS RETURNED BY `gettimeofday()`) OF COPYING A 32-BIT VARIABLE WITH EQUAL NUMBER OF BITS SET TO 0 AND 1, 1000 TIMES.

#	Description	Symbolic Analysis Results				Execution
		ANGR	*FuzzBall*	*S2E*	*Triton*	Time
	Original sample from Figure 5.	$✓_2$	$✓_2$	$✓_2$	$✓_2$	
	Deterministic Primitives					
0	Uses an integer counter as shown in Primitive 1.	$✗_k$	$✗_1$	$✗_{to}$	$✗_{364}$	0.00123
1	Same as *#0*, but with a floating point counter.	$✗_{to}$	$✗_1$	N/A	$✗_{367}$	0.00138
2	Uses a counter similar to Primitive 1, but raises a signal inside the loop and performs the increment inside the signal handler.	$✗_{to}$	$✗_1$	$✗_{to}$	$✗_{364}$	0.93016
3	Performs a simple unrolled bit-copy, similar to Primitive 2, where there is a unique copy for each bit in a byte and a loop iterates over all the bytes.	$✓_2$	$✗_1$	$✗_{to}$	$✗_{24}$	0.00009
4	Same as *#3*, but a single bit is copied at a time and a loop iterates over all bits.	$✓_2$	$✗_1$	$✗_{to}$	$✗_{24}$	0.00011
5	Performs a bit-copy using a signal as shown in Primitive 2.	$✗_{to}$	$✗_1$	$✗_{to}$	$✗_{24}$	0.04236
6	Writes the value to a file and immediately reads it back.	$✗_{\mathrm{pc}}$	$✗_1$	$✗_{to}$	$✗_1$	1.08820
	Non-Deterministic Primitives					
7	Thread-based timing (Primitive 8) of a trivial loop and majority logic, i.e. $repeat(\mathrm{loop/thread}, 3)$.	$✗_k$	$✗_{\mathrm{mmap}}$	$✗_{to}$	$✗_1$	7.18277
8	RTDSC-based timing of a trivial loop and majority logic, i.e. $repeat(\mathrm{loop/rdtsc}, 3)$.	$✗_{to}$	$✗_1$	$✗_{to}$	$✗_t$	0.06636
9	Primitive 4 with RTDSC-based timing and majority logic, i.e. $repeat(\mathrm{Primitive\ 4/rdtsc}, 3)$.	$✗_{sys}$	$✗_{\mathrm{uv}}$	$✗_{to}$	$✗_t$	0.98274
10	Same as *#9* but with thread-based timing.	$✗_{to}$	$✗_{\mathrm{mmap}}$	$✗_{to}$	$✗_t$	24.28911
11	Primitive 5 with RTDSC-based timing and majority logic, i.e. $repeat(\mathrm{Primitive\ 5/rdtsc}, 3)$.	$✗_{sys}$	$✗_1$	$✗_{255}$	$✗_t$	1761.31898
12	Same as *#11*, but with thread-based timing.	$✗_k$	$✗_{\mathrm{mmap}}$	$✗_{1252}$	$✗_t$	124.27554
13	Primitive 6 with RTDSC-based timing and majority logic, i.e. $repeat(\mathrm{Primitive\ 6/rdtsc}, 3)$.	$✗_0$	N/A	$✗_{255}$	$✗_t$	1184.09457
	Flow Combiners					
14	Composes *#0*, *#1*, and *#6*, i.e. $compose(\mathrm{counter/int}, \mathrm{counter/float}, \mathrm{file\_write})$.	$✗_k$	$✗_1$	$✗_{to}$	$✗_{242}$	0.97509
15	Randomly selects one of *#0*, *#1*, and *#6*, i.e. $select(\mathrm{counter\_int}, \mathrm{counter\_float}, \mathrm{file\_write})$.	$✗_{to}$	$✗_1$	$✗_{to}$	$✓_{244}$	0.00004
16	Combines *#2*, *#4*, and *#10* using majority logic, i.e. $majority(\mathrm{counter/signal}, \mathrm{Primitive\ 4/thread}, \mathrm{Primitive\ 2})$.	$✗_k$	$✗_{\mathrm{mmap}}$	$✗_{to}$	$✗_{258}$	8.91555
17	Combines *#1*, *#2*, *#4*, *#10*, and *#11* using majority logic, i.e. $majority(\mathrm{Primitive\ 4/thread}, \mathrm{Primitive\ 5/time}, \mathrm{bitcopy}, \mathrm{Primitive\ 2}, \mathrm{Primitive\ 1})$	$✗_{to}$	$✗_{\mathrm{mmap}}$	$✗_{to}$	$✗_{516}$	612.05139

6.1.2. Evaluation Platform. ANGR, Triton and S2E binaries were run on a machine with 1TB of RAM and 32 CPUs with 4 cores each. Fuzzball, due to issues installing it on the machine just described, was run in a virtual machine with 8 cores and 8GB of RAM. All ANGR samples were given a timeout of 3 hours, except sample 8 which was allowed to run for 27 hours. S2E was given a timeout of 8 hours and Triton 3 hours. All of these timeouts were significantly longer than the time cited to analyze samples with these engines in prior publications [31], [33], [29].

6.1.3. Evaluation Results. Table 1 shows the results of our tests. ANGR was able to fully analyze some of the deterministic obfuscations, but none of the non-deterministic ones. Of the deterministic samples, ANGR successfully detected two of the bit-copy samples and failed to detect the rest. It should be noted, though, that in our testing with a smaller symbolic input ANGR did successfully discover all paths for sample 0 and sample 1. As a result, we expect that if run longer ANGR would have been successful in identifying the implicit flows. The remainder of the deterministic samples failed for various reasons. Samples 1, 2, and 5 timed out and were manually killed. Sample 0 was killed by Linux because of resource exhaustion (no swap space or memory remained) and sample 6 caused the python interpreter to crash with a memory error. The non-deterministic and combined obfuscations failed for similar reasons. The only difference is that some used an unsupported system call that caused an error on all of ANGR's active paths.

Fuzzball was unable to completely explore any of the

Table 2. EFFECTS OF CLOCK PERTURBATION ON AN OBFUSCATED PROGRAM WITH OFFLINE TRAINING

Max Delay (ms)	Success Rate (%)	Tx Time (s)
0	99	0.948
5	100	1.210
10	99	1.493
15	0	1.751
20	0	2.067
25	0	2.316
30	0	2.600
35	0	2.876
40	0	3.287
45	0	3.503
50	0	3.928

Table 3. EFFECTS OF CLOCK PERTURBATION ON AN OBFUSCATED PROGRAM WITH STARTUP TRAINING

Max Delay (ms)	Success Rate (%)	Tx Time (s)
0	95	0.737
5	100	1.214
10	99	2.326
15	92	3.680
20	99	5.090
25	99	5.894
30	98	7.893
35	100	8.071
40	100	8.619
45	99	8.744
50	97	9.486

obfuscated samples. In most cases, Fuzzball found the path to the failure branch, which is equivalent to running the software. The remaining samples caused Fuzzball to produce an error. All of the samples that required multiple threads failed due to an `mmap` operation that could not be handled by Fuzzball; however, since `mmap` is required to use the `pthreads` library, we could not remove `mmap` from the sample. The only sample remaining is sample 9, which caused Fuzzball to fail because it found an undeclared variable in the disassembled code. We were not sure how to fix this error and counted it as a failure.

S2E was configured to use a depth-first search strategy to discover paths that maximize code coverage. However, S2E was not able to find all viable paths in any of the samples. As shown in Table 1, S2E produced many concrete counter examples (ranging from 255 to 1252) that were supposed to trigger different execution paths, but which only triggered one of the target paths in the obfuscated sample, leaving the other undiscovered. In addition, sample 1 triggered an error in the analysis that caused S2E to abort and all other samples timed out. Hence, S2E failed on all the samples.

Except for sample 15, Triton's behavior was similar to S2E's: it reported false positive alternative inputs that would only trigger one of the target paths in the obfuscated sample. Moreover, for samples 8 through 13, Triton timed out and did not produce any results; sample 15 was an exception as the tool was able to successfully discover both possible execution paths while also reporting about 244 false positive alternative counterexamples.

6.2. Covert Channel Detection Techniques

To evaluate the stealthiness of our obfuscations against analyses aimed at detecting covert channels, we considered two representative kinds of analyses: (i) clock-perturbation techniques that aim to identify timing channels; and (ii) analyses that examine the system calls executed by a program for indications of suspicious or anomalous behavior.

6.2.1. Clock Perturbation Attacks. The idea behind clock perturbation (or "clock fuzzing") is to reduce the bandwidth of timing covert channels by introducing noise into system clocks[34], [35]. This noise causes the accuracy of time measurements to decrease system wide, meaning timing channels must have delays larger than the noise since those near or less than the maximum noise value will generate random results. In our context, clocks are simulated by racing "fast" and "slow" threads, which means perturbations to the system clock would be manifested by corresponding changes in the observed execution time for these threads. Based on this observation, we simulated clock perturbation by adding independent random delays to the transmission and training threads in our obfuscated programs. These delays inject a random perturbation to each thread's execution time; this can, for example, cause the fast thread to run slower than the slow thread if the delay added to the former is significantly larger than the latter.

We tested clock perturbation against two obfuscated programs: one with offline training and one with startup training. In both cases we set the training confidence to 99% and varied the maximum delay and averaged the success rate and secret value transmission time across 100 runs. It should be noted that despite training to 99%, perturbations after the training ends can cause failures since we have not implemented on-demand training, leading to lower success rates in some cases. As Table 2 shows, with offline training, clock perturbation quickly reduces the accuracy of the covert channel as the induced delay increases since the timing is increasingly influenced by randomness. Startup training performs much better since the training dynamically determines an appropriate transmission speed, as shown in table 3. Clock propagation therefore reduces the bandwidth of the channel, as expected, but does not defeat the obfuscation.

6.2.2. System Call-based Anomaly Detection. There have been a number of proposals for anomaly detection by examining the system call behavior of programs (e.g., see [36], [37]). The usual application of this approach involves comparing the system calls executed by a program against a model of its "normal" behavior. However, in a program that uses covert channels for obfuscation, the normal behavior includes the system calls used by the covert channels, so the question to be addressed is: does the use of covert channels result in unusual system call behaviors? To address this question, we examined the system calls executed by each obfuscated program using the *strace* utility and compared them with those executed by the corresponding unobfuscated program; the difference between the two would be the system calls introduced by the covert channel obfuscation.

In our experiments, we found that if the obfuscation used alarms, threads, file I/O or JIT, the system call behavior would be drastically different. If alarms were used, one would see many `rt_sigreturn` and `tgkill` calls, which would likely be flagged as anomalous.

The threaded samples contained many calls to the `clone` system call due to the fact that every time we leak a bit, two new threads are created. While this is likely to be marked as anomalous, implicit timing can be performed with only one additional thread (as mentioned in section 4.4). When the single thread timing is applied to one of the obfuscations, the resulting system call behavior is identical to the original program to which a single dummy thread had been added. Consequently, a system-call-based detector would mark all multi-threaded programs as anomalous.

The file I/O channels would likely be suspicious since the obfuscations read and write to the same file. The file cache channel, however, can also be performed by purely using file reads which is much less suspicious [38].

Finally, the JIT based channel has many `brk` system calls, due to the implementation of the JIT translator built into the obfuscator. Consequently, this would not detect the *obfuscation*, but the presence of JIT translation. Since JIT compilers are often included in interpreters such as the Java Virtual Machine (JVM), it is unlikely that anomaly detection could be used to detect the JIT based obfuscation in general.

These results show the flexibility of our obfuscation as the parts that make up the covert channel can be swapped out to maximize stealth.

6.3. Performance

To investigate the relative performance of the primitives proposed here we obfuscated a 32-bit assignment `a = b` with each of the transformations from Table 1. We report cumulative wall-clock times of performing each assignment 1000 times. Timings were collected on a laptop with a 2.9GHz Intel Core i7 with 16GB of main memory and 2TB of SSD disk, running MacOS 10.12.3.

As expected, the transformations based on JITting and file caching are much more expensive than the other transformations. It should be noted that even such expensive obfuscations can be useful under the right circumstances: to defeat symbolic analysis it can be enough to apply a transformation to one or a few strategic assignments in the program, and to avoid those that are potential hotspots.

7. Related Work

The work most closely related to ours is that on side-channel-based information leakage and various covert-channel attacks. Also related are works on information flow tracking and symbolic execution. The idea of probabilistic obfuscation was also mentioned in [39] where authors propose a technique to obfuscate the control flow of the program such that it is no longer deterministic.

7.1. Covert Channels and Side Channels

Covert channels and side channels are information channels that use properties of a computation that are distinct from the actual computation to propagate information. The distinction between these terms is typically one of intent—"covert channel" usually denotes deliberate use of a channel to transmit information, while "side channel" refers to inadvertent information transmission.

There is a significant body of research on side-channel attacks that attempt to extract keys from cryptographic code [40]. A variety of attacks have been proposed in the literature, based on different kinds of observations, e.g., timing characteristics [23], energy usage [24], and cache hit/miss behavior [38], [41], [42], [43]; Biswas *et al.* give a survey [44]. In such scenarios, the programs under attack—typically, cryptographic codes—are not deliberately engineered to yield the information the attacker is trying to extract. Correspondingly, the defenses proposed against such information leakage typically focus on "crypto-like" code: small fragments with limited and tightly controlled interactions with external code such as libraries, runtime system, or the operating system [45], [46], [47]. Importantly, the tightly controlled nature of such software, and the fact that side-channel information leakage is not deliberate, limits the side channels that are usefully exploitable.

Such channels can also be deliberately engineered to leak information. In this case, there may be a wide variety of covert channels potentially available for use by the program, and they may be exploited in different ways [6], [48], [2], [3]. Such mechanisms for information exfiltration represent an emerging class of malicious behavior that are not handled by existing analysis frameworks.

7.2. Information Flow Tracking

Information flow tracking systems try to enforce information flow policies which are derived by the confidentiality rules. *Security typed languages*, in which the types are augmented with security labels to specify (and enforce) information flow policies, provide strong guarantees towards secure information flow. Sabelfeld [49] gives a survey on securely-typed language approaches. These approaches, however, are too strict and can potentially lead to too many false positives that render them inapplicable [21]. Moreover, these approaches are applicable in the context of protecting secrets from being observed by an attacker (e.g., cryptographic keys) and require source code. The focus of this paper, however, is in the context of analyzing binary code where users run untrusted applications that most likely have access to their personal data (e.g., on mobile phones). In these situations, securely typed language approaches do not help to protect the confidentiality of the data [50].

To address these problems, researches have proposed approaches that mark and track sensitive data (i.e., *tainted* *data*) and prevent the tainted data from being leaked [51], [52], [53], [54], [55]. An important shortcoming of such dynamic taint propagation approaches is their inability to track implicit information flows [17] and imprecise results in the presence of code obfuscation techniques [56], [57].

7.3. Symbolic Execution

Symbolic execution is used in a wide variety of security-related analyses [58], [59], [60], [1], [61]. While capable of sophisticated reasoning about program behavior, symbolic execution suffers from a number of practical drawbacks, including path explosion, dealing with indirect references, and memory modeling and system calls [62], [58]. Symbolic execution systems can also have trouble in reasoning about obfuscated code [63], [57], [22], resulting in significant degradation in performance and precision. Similar to the previous studies, this work helps researchers better understand the shortcomings of symbolic execution in dealing with obfuscated code.

Other researchers have used cryptographic hash functions to hide the relationship between branch points and input values in the code [64], [65]. While effective in subverting symbolic execution from accurately determining different execution paths in the program, the use of cryptographic functions raises suspicions in detection mechanisms. Moreover, these studies target the limits of the underlying SMT solvers and are useful in determining the theoretical boundaries of symbolic execution analysis.

8. Conclusion

This paper describes a new obfuscation technique that exploits covert channels, arising from a program's runtime interactions with its execution environment, to obfuscate information flows and make them harder to track. Unlike existing obfuscation techniques, our approach removes information flows from the program's code, rerouting them through the runtime system and/or operating system and thereby rendering them invisible to conventional program analyses. The work is also motivated by the need to understand the foundations of emergent techniques for sidestepping privacy protections on mobile devices and exfiltrating sensitive information. We describe a semantic framework for covert-channel-based information propagation, show how covert channels can be used as a code obfuscation technique, and introduce the notion of probabilistic obfuscation. Experimental evaluation of a prototype implementation of our ideas shows that our obfuscation is stealthy, successfully evades state-of-the-art information flow analysis tools, and is robust against clock-fuzzing and system-call tracing analyses aimed at detecting covert channels.

Acknowledgments

This research was supported in part by the National Science Foundation (NSF) under grant CNS-1525820.

References

[1] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *Proc. 2015 IEEE Symposium on Security and Privacy*, 2015, pp. 674–691.

[2] A. Al-Haiqi, M. Ismail, and R. Nordin, "A new sensors-based covert channel on android," *The Scientific World Journal*, vol. 2014, 2014.

[3] J.-F. Lalande and S. Wendzel, "Hiding privacy leaks in android applications using low-attention raising covert channels," in *Availability, Reliability and Security (ARES), 2013 Eighth International Conference on*. IEEE, 2013, pp. 701–710.

[4] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM, 2012, pp. 51–60.

[5] E. Owusu, J. Han, S. Das, A. Perrig, and J. Zhang, "Accessory: password inference using accelerometers on smartphones," in *Proceedings of the Twelfth Workshop on Mobile Computing Systems & Applications*. ACM, 2012, p. 9.

[6] R. Schlegel, K. Zhang, X.-y. Zhou, M. Intwala, A. Kapadia, and X. Wang, "Soundcomber: a stealthy and context-aware sound trojan for smartphones." in *NDSS*, vol. 11, 2011, pp. 17–33.

[7] C. Wang, J. Hill, J. C. Knight, and J. W. Davidson, "Protection of software-based survivability mechanisms," in *Dependable Systems and Networks (DSN)*, 2001, pp. 193–202.

[8] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proc. 10th ACM conference on Computer and communications security*, 2003, pp. 290–299.

[9] S. Chow, P. Eisen, H. Johnson, and P. van Oorschot, "White-box cryptography and an AES implementation," in *9th Annual Workshop on Selected Areas in Cryptography (sac 2002.*, 2002.

[10] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *Proc. 8th Int. Conf. on Information Security Applications*, 2007, pp. 61–75.

[11] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs (extended abstract)," in *Advances in Cryptology - CRYPTO 2001*, 2001, lNCS 2139.

[12] B. Barak, "Hopes, fears, and software obfuscation," *Commun. ACM*, vol. 59, no. 3, pp. 88–96, Feb. 2016.

[13] O. Chen, C. Meadows, and G. Trivedi, "Stealthy protocols: Metrics and open problems," in *Concurrency, Security, and Puzzles*. Springer International Publishing, 2017, pp. 1–17.

[14] V. Crespi, G. Cybenko, and A. Giani, "Engineering statistical behaviors for attacking and defending covert channels," *IEEE Journal of Selected Topics in Signal Processing*, vol. 7, no. 1, pp. 124–136, 2013.

[15] C. Collberg and J. Nagra, "Surreptitious software," *Upper Saddle River, NJ: Addision-Wesley Professional*, 2010.

[16] L. Cavallaro, P. Saxena, and R. Sekar, "On the limits of information flow techniques for malware analysis and containment," in *Proceedings of the 5th International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, ser. DIMVA '08. Springer-Verlag, 2008, pp. 143–163.

[17] ——, "Anti-taint-analysis: Practical evasion techniques against information flow based malware defense," Tech. Rep., 2007.

[18] G. Sarwar, O. Mehani, R. Boreli, and M. A. Kaafar, "On the effectiveness of dynamic taint analysis for protecting against private information leaks on Android-based devices." in *SECRYPT*, 2013, pp. 461–468.

[19] A. Russo, A. Sabelfeld, and K. Li, "Implicit flows in malicious and nonmalicious code," in *Logics and Languages for Reliability and Security*, 2010, pp. 301–322.

[20] Y. Liu and A. Milanova, "Static information flow analysis with handling of implicit flows and a study on effects of implicit flows vs explicit flows," in *Proc. 2010 14th European Conference on Software Maintenance and Reengineering (CSMR)*, March 2010, pp. 146–155.

[21] D. King, B. Hicks, M. Hicks, and T. Jaeger, "Implicit flows: Can't live with 'em, can't live without 'em," in *Proceedings of the International Conference on Information Systems Security (ICISS)*, R. Sekar and A. K. Pujari, Eds. Springer, 2008.

[22] B. Yadegari, J. Stephens, and S. Debray, "Analysis of exception-based control transfers," in *Proceedings of the 7th ACM Conference on Data and Application Security and Privacy (CODASPY)*, 2017.

[23] P. C. Kocher, "Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems," *Lecture Notes in Computer Science*, vol. 1109, pp. 104–113, 1996.

[24] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," *Lecture Notes in Computer Science*, vol. 1666, pp. 388–397, 1999.

[25] Y. Yarom and K. Falkner, "Flush+ reload: A high resolution, low noise, l3 cache side-channel attack." in *USENIX Security*, vol. 2014, 2014, pp. 719–732.

[26] H.-J. Boehm and M. Weiser, "Garbage collection in an uncooperative environment," *Softw. Pract. Exper.*, vol. 18, no. 9, pp. 807–820, Sep. 1988. [Online]. Available: http://dx.doi.org/10.1002/spe.4380180902

[27] T. Zhang, C. Jung, and D. Lee, "Prorace: Practical data race detection for production use," *SIGOPS Oper. Syst. Rev.*, vol. 51, no. 2, pp. 149–162, Apr. 2017.

[28] E. B. Wilson, "Probable inference, the law of succession, and statistical inference," *Journal of the American Statistical Association*, vol. 22, no. 158, pp. 209–212, 1927.

[29] V. Chipounov, V. Kuznetsov, and G. Candea, "S2e: a platform for in-vivo multi-path analysis of software systems," in *Proc. 15th Int. Conf. on Architectural Support for Programming Languages and Operating Systems*, Mar. 2011, pp. 265–278.

[30] D. Babić, L. Martignoni, S. McCamant, and D. Song, "Statically-directed dynamic automated test generation," in *Proc. 2011 Int. Symp. on Software Testing and Analysis*, 2011, pp. 12–22.

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

[32] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France*, 2015, pp. 31–54.

[33] J. Salwan, R. Thomas, and A. Guinet, "Playing with the tigress," 2016, https://github.com/JonathanSalwan/Tigress_protection.

[34] W.-M. Hu, "Reducing timing channels with fuzzy time," *Journal of Computer Security*, vol. 1, no. 3-4, pp. 233–254, 1992.

[35] P. A. Karger, M. E. Zurko, D. W. Bonin, A. H. Mason, and C. E. Kahn, "A retrospective on the VAX VMM security kernel," *IEEE Transactions on Software Engineering*, vol. 17, no. 11, pp. 1147–1165, 1991.

[36] D. Mutz, F. Valeur, G. Vigna, and C. Kruegel, "Anomalous system call detection," *ACM Trans. Inf. Syst. Secur.*, vol. 9, no. 1, pp. 61–93, Feb. 2006.

[37] A. Frossi, F. Maggi, G. L. Rizzo, and S. Zanero, "Selecting and improving system call models for anomaly detection." in *DIMVA*. Springer, 2009, pp. 206–223.

[38] C. Percival, "Cache missing for fun and profit," 2005.

[39] A. Pawlowski, M. Contag, and T. Holz, "Probfuscation: An obfuscation approach using probabilistic control flows," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 165–185.

[40] F.-X. Standaert, T. G. Malkin, and M. Yung, "A unified framework for the analysis of side-channel key recovery attacks," in *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 2009, pp. 443–461.

[41] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-vm side channels and their use to extract private keys," in *Proceedings of the 2012 ACM conference on Computer and communications security*. ACM, 2012, pp. 305–316.

[42] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: the case of aes," in *Cryptographers' Track at the RSA Conference*. Springer, 2006, pp. 1–20.

[43] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM conference on Computer and communications security*. ACM, 2009, pp. 199–212.

[44] A. K. Biswas, D. Ghosal, and S. Nagaraja, "A survey of timing channels and countermeasures," *ACM Comput. Surv.*, vol. 50, no. 1, pp. 6:1–6:39, Mar. 2017.

[45] D. J. Bernstein, T. Lange, and P. Schwabe, "The security impact of a new cryptographic library," in *International Conference on Cryptology and Information Security in Latin America*. Springer, 2012, pp. 159–176.

[46] D. Molnar, M. Piotrowski, D. Schultz, and D. Wagner, "The program counter security model: Automatic detection and removal of control-flow side channel attacks," in *International Conference on Information Security and Cryptology*. Springer, 2005, pp. 156–168.

[47] J. B. Almeida, M. Barbosa, G. Barthe, F. Dupressoir, and M. Emmi, "Verifying constant-time implementations," in *25th USENIX Security Symposium*, 2016, pp. 53–70.

[48] C. Marforio, H. Ritzdorf, A. Francillon, and S. Capkun, "Analysis of the communication between colluding applications on modern smartphones," in *Proceedings of the 28th Annual Computer Security Applications Conference*, ser. ACSAC '12, 2012, pp. 51–60.

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

[50] L. Cavallaro, P. Saxena, and R. Sekar, "Anti-taint-analysis: Practical evasion techniques against information flow based malware defense," *Secure Systems Lab at Stony Brook University, Tech. Rep*, pp. 1–18, 2007.

[51] W. Enck, P. Gilbert, S. Han, V. Tendulkar, B.-G. Chun, L. P. Cox, J. Jung, P. McDaniel, and A. N. Sheth, "Taintdroid: an information-flow tracking system for realtime privacy monitoring on smartphones," *ACM Trans. Comp. Sys. (TOCS)*, vol. 32, no. 2, p. 5, 2014.

[52] S. Arzt, S. Rasthofer, C. Fritz, E. Bodden, A. Bartel, J. Klein, Y. Le Traon, D. Octeau, and P. McDaniel, "Flowdroid: Precise context, flow, field, object-sensitive and lifecycle-aware taint analysis for android apps," *ACM Sigplan Notices*, vol. 49, no. 6, pp. 259–269.

[53] M. G. Kang, S. McCamant, P. Poosankam, and D. Song, "Dta++: Dynamic taint analysis with targeted control-flow propagation." in *NDSS*, 2011.

[54] P. Vogt, F. Nentwich, N. Jovanovic, E. Kirda, C. Kruegel, and G. Vigna, "Cross site scripting prevention with dynamic data tainting and static analysis." in *NDSS*, vol. 2007, 2007, p. 12.

[55] J. Clause, W. Li, and A. Orso, "Dytan: a generic dynamic taint analysis framework," in *Proceedings of the 2007 international symposium on Software testing and analysis*. ACM, 2007, pp. 196–206.

[56] B. Yadegari and S. Debray, "Bit-level taint analysis," in *Source Code Analysis and Manipulation (SCAM), 2014 IEEE 14th International Working Conference on*. IEEE, 2014, pp. 255–264.

[57] ——, "Symbolic execution of obfuscated code," in *Proc. 22nd ACM Conference on Computer and Communications Security*, 2015, pp. 732–744.

[58] E. J. Schwartz, T. Avgerinos, and D. Brumley, "All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask)," in *Security and privacy (SP), 2010 IEEE symposium on*. IEEE, 2010, pp. 317–331.

[59] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 14th ACM conference on Computer and communications security*. ACM, 2007, pp. 116–127.

[60] A. Moser, C. Kruegel, and E. Kirda, "Exploring multiple execution paths for malware analysis," in *Security and Privacy, 2007. SP'07. IEEE Symposium on*. IEEE, 2007, pp. 231–245.

[61] J. Qiu, B. Yadegari, B. Johannesmeyer, S. Debray, and X. Su, "Identifying and understanding self-checksumming defenses in software," in *Proceedings of the 5th ACM Conference on Data and Application Security and Privacy*. ACM, 2015, pp. 207–218.

[62] C. Cadar and K. Sen, "Symbolic execution for software testing: three decades later," *Comm. ACM*, vol. 56, no. 2, pp. 82–90, 2013.

[63] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 189–200.

[64] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 210–226.

[65] M. Sharif, A. Lanzi, J. Giffin, and W. Lee, "Impeding malware analysis using conditional code obfuscation," in *15th Annual Network and Distributed System Security Symposium (NDSS)*, Feb. 2008.