ConcSpectre: Be Aware of Forthcoming Malware Hidden in Concurrent Programs

Yang Liu
Xi'an Jiaotong University
Xi'an, Shaanxi, China
yangliu@xjtu.edu.cn

Ming Fan
Xi'an Jiaotong University
Xi'an, Shaanxi, China
mingfan@xjtu.edu.cn

Ting Liu
Xi'an Jiaotong University
Xi'an, Shaanxi, China
tingliu@mail.xjtu.edu.cn

Yu Hao
University of California Riverside
CA, USA
yhao016@ucr.edu

Zisen Xu
Xi'an Jiaotong University
Xi'an, Shaanxi, China
xzs05350332@stu.xjtu.edu.cn

Kai Chen
Chinese Academy of Sciences
Beijing, China
chenkai@iee.ac.cn

Hao Chen
University of California, Davis
CA, USA
chen@ucdavis.edu

Yan Cai
Chinese Academy of Sciences
Beijing, China
ycai.mail@gmail.com

***Abstract*—Concurrent programs with multiple threads executing in parallel are widely used to unleash the power of multi-core computing systems. Due to their complexity, large amounts of researches focus on testing and debugging concurrent programs. Besides correctness, we find that security can also be compromised by concurrency. In this paper, we present concurrent program spectre (ConcSpectre), a new security threat that hides malware in non-deterministic thread interleavings. To demonstrate such threat, we have developed a stealth malware technique called Concurrent Logic Bomb (CLB) by partitioning a piece of malicious code and injecting its components separately into a concurrent program. The malicious behavior can be triggered by certain thread interleavings that rarely happen (e.g., $<$1%) under a normal execution environment. However, with a new technique called Controllable Probabilistic Activation (CPA), we can activate such ConcSpectre malware with a very high probability (e.g., $>$90%) by remotely disturbing thread scheduling. In the evaluation, more than 1,000 ConcSpectre samples are generated, which bypassed most of the anti-virus engines in VirusTotal and four well-known online dynamic malware analysis systems. We also demonstrate how to remotely trigger a ConcSpectre sample on a web server and control its activation probability. Our work shows an urgent need for new malware analysis methods for concurrent programs. All source code and analysis results are available on https://git.io/CLB.**

***Index Terms*—Concurrent Programs; Software Security; ConcSpectre; Concurrent Logic Bomb; Controllable Probabilistic Activation**

I. Introduction

Many strategies have been implemented to unleash the full potential of modern processors, such as out-of-order execution, branch prediction and speculative execution strategies. These optimization technologies significantly enhance the performance but at the same time dramatically increase the complexity of the hardware systems. Complexity may introduce security risks such as Meltdown [1] and Spectre [2]. Meltdown utilizes side effects of out-of-order execution to read arbitrary kernel-memory locations, including personal data and passwords. Spectre attacks involve inducing a victim to speculatively perform operations that leak the victim's confidential information via a side channel to the adversary. The community was astonished by these discoveries due to its severity, as these optimization technologies have been used for decades.

In this paper, we present a new threat to hide malware in concurrent programs. Concurrent programs with multiple threads executing in parallel are widely used to improve system efficiency. Meanwhile, the inherent non-determinism of thread interleavings also significantly increases the complexity of programs. In general, the number of possible interleavings of a concurrent program with n threads each executing k steps can be as large as $(nk)!/(k!)^n \geq (n!)^k$ [3], a number that is double exponential to both n and k. Since it is impossible to check all interleavings of a nontrivial program, a piece of malware triggered only by several specific thread interleavings would be extremely difficult to detect. Similar to Spectre, this threat also exploits the complexity of the concurrency mechanism to cover its malicious behavior, which is used to unleash the power of modern multi-core computing systems. We name it concurrent program spectre (ConcSpectre).

Current malware detection techniques mainly rely on static malicious signatures and dynamic analysis results [4]. However, the static signatures can be easily changed using obfuscation techniques. Dynamic analysis technologies aim to execute each execution path to trigger the malicious behavior for further verification. However, exploring all paths is extremely hard in the era of multi-core processors due to the non-determinism of thread interleavings.

Consider the multi-threaded program given in Fig. 1. The main thread creates two threads $T1$ and $T2$ to concurrently execute the function *multi_download* to download four files in

This work was partially supported by National Key R&D Program of China (2018YFB0803501), National Natural Science Foundation of China (U1766215, 61772408, 61632015, 61833015, 62002281, 61902306), the Fundamental Research Funds for the Central Universities, China Postdoctoral Science Foundation (2019TQ0251, 2020M683520, 2020M673439), the 2020 Industrial Internet Innovation Development Project - Industrial Internet Penetration Testing and Crowdsourced Testing Platform (No. TC200H01P), the Youth Talent Support Plan of Xi'an Association for Science and Technology (095920201303), CCF-Tencent Open Research Fund. Corresponding author: Ming Fan.

```
void multi_download(int N) {
    for(int i = 0; i < N; i++) {
        LOCK;
         if(!is_download(i)) {
             set_download(i);
             download(i); }
        UNLOCK; }
}
void main(int N){
    CREATE(T1,multi_download,4);
    CREATE(T2,multi_download,4); }
```

Fig. 1: Code snippet of a multi-threaded download program.

total. The functions *is_download* and *set_download* are used to avoid downloading the same file more than once. A typical execution trace is $\pi1$: T1(*download*(0)) - T2(*download*(1)) - T1(*download*(2)) - T2(*download*(3)). However, $\pi1$ is not the only trace under input $N = 4$. For example, if there is a congestion during the execution of *download*(0), a different trace can be $\pi2$: T1(*download*(0)) - T2(*download*(1)) - T2(*download*(2)) - T1(*download*(3)). A malware analysis on $\pi1$ and $\pi2$ may report different results. The behavior of a concurrent program relies on not only its inputs but also the thread scheduling. The inherent non-determinism of multi-threaded executions invalidates the assumption of deterministic behavior under fixed inputs and thus exhibits a threat to the current malware detection techniques, which is the intuition of our ConcSpectre.

In order to implement ConcSpectre by exploiting concurrency, there are two main challenges. The first is to snugly hide a malware sample in a concurrent program while bypassing the malicious behavior detection of modern malware detection tools. The second is to trigger the malicious behavior effectively in a controllable manner. This paper addresses these challenges with a new stealth malware technique called Concurrent Logic Bomb (CLB) and a new activation technique called Controllable Probabilistic Activation (CPA).

The idea of the CLB technique is to partition a piece of malware and inject its components into many concurrently executed program fragments such that (1) each individual component is benign so the malware detection tool does not raise alarms when monitoring its execution, (2) there exist specific orderings of the components that trigger the malicious behavior, and (3) malicious behaviors are well-hidden from typical executions. There are different strategies to partition malware, identify the injection locations in a host program and arrange the orders to manifest malicious behavior. Therefore, a piece of malware processed by the CLB technique may yield multiple pieces before injecting into a concurrent program. To demonstrate the feasibility of the CLB technique with a real case, we partition the malware sample *BullMoose* and inject its components into various locations of several publicly available concurrent programs. More than 1,000 malware samples have been generated, which evade the detection of most of the anti-virus engines in VirusTotal [5], as well as four well-known online dynamic malware analysis systems.

For each piece of CLB malware, there exist certain orderings that can trigger the malicious behavior. These orderings rarely happen during normal execution conditions. The idea of CPA is to disturb the normal execution condition such that the orderings that trigger malicious behavior are no longer rare. In the experiments, we implement a CPA technique based on the system load, and demonstrate that this CPA technique can significantly increase the probability of the rare orderings with a group of real attacks. By combining CLB and CPA techniques, attackers can control the activation probability of ConcSpectre malware, which is less than 1% under normal execution environment and higher than 90% under attack.

In summary, this paper makes the following contributions:

- We reveal a new security threat called ConcSpectre to concurrent programs, which calls for an urgent redesign of malware detection techniques for concurrent programs to prevent forthcoming threats.
- We propose a new stealth malware technique called CLB to hide a malware sample into concurrent programs. Leveraging the difficulty of analyzing the interleaving of different threads, the concealed malware can evade all the state-of-the-art dynamic and static detectors.
- We design a new malware activating approach called CPA to trigger ConcSpectre malware based on the system workload. CPA can drastically increase the probability of triggering malicious behavior that is stealthy under normal execution conditions.

II. Overview

A. Motivating Example

We use the programs given in Fig. 2 as a running example to explain the basic idea of ConcSpectre. As shown in Fig. 2(a), a snippet of malware calls *get_data*() and *send_data*() to retrieve sensitive data and transmit it. A dynamic malware detection tool can report this illegal activity when it monitors the execution of the malware since sensitive data are retrieved and then transmitted.

As shown in Fig. 2(b), we transform the program to *malware_C*() where the order of the calls to *get_data*() (Line 19) and *send_data*() (Line 14) are reversed and separated into two LOCK/UNLOCK components (S1 and S2). An additional variable *order* is inserted for controlling the execution. Monitoring the execution of *malware_C*() by running it within a single thread does not raise any alarms since no sensitive data are being retrieved first and then transmitted.

Consider the host program given in Fig. 2(c) where two threads ($T1$ and $T2$) simultaneously invoke *malware_C*() in *func*(). We show that with input $x = 1$, the malicious behavior of data-stealing can happen under specific interleavings between the two threads. Fig. 3 lists six execution traces that cover all possible combinations of the two branches between the two threads, where (S1,T) indicates the if-statement in S1 executes with a true branch, and (S2,F) means S2 executes with a false branch. While *get_data*() is invoked in all execution traces, *send_data*() is only invoked in $\pi1$ and $\pi4$. It can be observed that the malicious behavior manifests in

```c
void malware() {
    LOCK(mutex);
    get_data();
    send_data();
    UNLOCK(mutex);
}

void get_data();
void send_data();
```

(a) A snippet of malware

```c
void malware_C() {
    LOCK(mutex);              //S1
    if(order == 1) {          //S1
        order = 2;            //S1
        send_data();}         //S1
    UNLOCK(mutex);            //S1
    LOCK(mutex);              //S2
    if(order == 0) {          //S2
        order = 1;            //S2
        get_data();}          //S2
    UNLOCK(mutex);            //S2
}
```

(b) ConcSpectre Malware

```c
void host_C(int x){
    if(x == 1) {
        order = 0;
        CREATE(T1,func);
        CREATE(T2,func);
    }
}
void func() {
    func_original();
    malware_C();
}
```

(c) Host program of ConcSpectre

Fig. 2: A demo case of ConcSpectre.

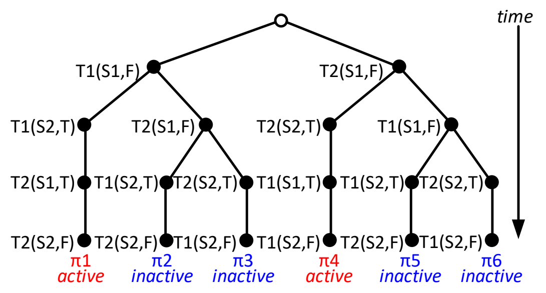


Fig. 3: Execution traces of ConcSpectre malware in Fig. 2. The last row indicates whether the malicious behavior is activated.

$\pi 1$ when $T2$ transmits the data obtained by $T1$, and in $\pi 4$ when $T1$ transmits it obtained by $T2$. That is, by monitoring an execution, the malware detection tool has a probability of 1/3 to detect the malicious behavior. This seems not bad, but the number of interleavings can increase drastically and the probability can decrease sharply when the number of malware components and host program's threads increases. For example, there are 1,680 possible interleavings when there are three malicious components injected into three parallel threads. If the trigger condition of malware is their execution and injected orders are fully reversed, only 5.7% of all possible interleavings can trigger the malicious behavior. The number becomes 63,063,000 with 0.07% activating malicious behavior when there are four components and four threads. It shows that the chance of detecting ConcSpectre malware diminishes when the malware sample or the host concurrent program is nontrivial.

ConcSpectre can be exploited in at least two scenarios. Firstly, it can be used to launch advanced persistent threat attacks against high-security targets. The ConcSpectre may hide malware in some large concurrent software to bypass the rigorous security reviews in these high-security systems, even when the source code is open to the security analyst. Secondly, ConcSpectre can be applied to launch various large-scale attacks, such as Botnet and worm. Specifically, ConcSpectre zombies in a Botnet could hide their abnormal behavior well by randomly activating once in thousands of runs.

B. Basic Assumptions

The work in this paper is built on the following assumptions.

Firstly, a piece of malware can be partitioned and each component is not detectable by current malware analysis techniques. Since each component by itself does not cause any harm, its behavior is usually not suspicious. In Sec. V-C, we partition four real malware samples into many components and no malware detection engines raise alarms.

Secondly, ConcSpectre malware can be installed on a victim's system that supports multithreading using various methods (e.g., fishing, social engineering, etc.). Then, ConcSpectre can hide malicious code, bypass current malware detection, and probabilistically activate the malware. In Sections IV and V, we demonstrate how to inject malware into programs in common concurrency benchmarks. The generated ConcSpectre malware can bypass current malware detection tools.

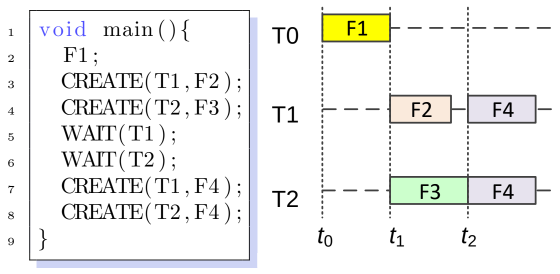
Thirdly, the attacker can influence the thread scheduling of the victim's system. This assumption is reasonable since we do not require precise thread scheduling. Specifically, attackers can perturb thread switching by sending suspend commands, increasing the system load, etc. In Sec. V, we demonstrate how to remotely activate a ConcSpectre malware on a web server by increasing the workload on the target machine.

III. Concurrent Logic Bomb

CLB is a technique for hiding malware by partitioning a piece of malware and hiding its components into a concurrent program. In this paper, we partition a malware sample manually with the following consideration: (1) automated code partition has been a difficult problem for decades; (2) with domain and code knowledge, an attacker may give a partition trickier than any automated approach. In this section, we will first focus on automatically finding suitable locations to inject partitioned malware sections, and then confirm the stealthiness of the proposed CLB technique.

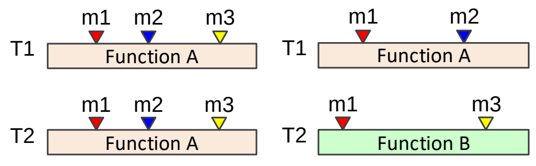
A. Malware Injection

The CLB technique injects partitioned malware components into different functions of a concurrent program. To choose the right hosting functions, we classify the functions in a concurrent program into three types, as illustrated in Fig. 4:
Non-parallel-execution (NPE) functions cannot be executed with any other functions simultaneously (e.g., F1).
Cross-parallel-execution (CPE) functions can be executed



```
void main(){
  F1;
  CREATE(T1,F2);
  CREATE(T2,F3);
  WAIT(T1);
  WAIT(T2);
  CREATE(T1,F4);
  CREATE(T2,F4);
}
```

Fig. 4: Three types of functions in concurrent programs.



(a) SPE **(b)** CPE

Fig. 5: Malware section injection methods.

with other functions concurrently (e.g., F2 and F3).

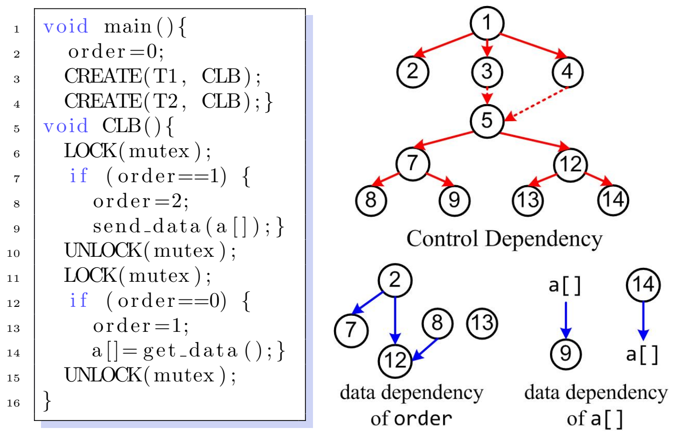
Self-parallel-execution (SPE) functions can be executed by multiple threads concurrently, but cannot be executed with other functions in parallel (e.g., F4).

Both code analysis and execution monitoring can be applied to identify the aforementioned three types of functions. Apparently, NPE functions are not good host candidates because their executions are affected by thread interleaving indirectly through CPE and SPE functions.

Assuming that three *malware components* ($m1$: stealing and saving sensitive data, $m2$: sending data, and $m3$: releasing data) are injected into an SPE function, as shown in Fig. 5(a), two types of faults may occur: (1) *Repeated execution*. If $m3$ has been executed in Thread 1, its re-execution in Thread 2 is erroneous. (2) *Execution with wrong order*. If $m2$ in Thread 2 sends the sensitive data that has been cleared by $m3$ in Thread 1, its execution is erroneous.

Therefore, we have to design a control module to ensure the *malware components* are executed correctly. One approach is to create a shared variable to indicate whether a *malware component* can be executed. After one component is executed, the variable is set to the value representing the next component. In Fig. 2, we adopt this approach by using the variable `order`. Of course, other strategies can also be used, such as backward setting and forward searching. In backward setting, the current component can turn on the execution permission of the next component while turning off others. In forward searching, the current component has to confirm whether certain other components have been executed successfully before its execution. We define a *malware component* with its control module as a *malware section* that is a basic unit in a piece of ConcSpectre malware. Although control modules introduce additional dependency among *malware sections*, such type of dependency cannot be exploited by malware analysis tools. We will discuss this in Sec. III-B.

When we inject *malware components* into *CPE* functions, the injection positions of all sections are different. As shown



```
1  void main(){
2    order=0;
3    CREATE(T1, CLB);
4    CREATE(T2, CLB);}
5  void CLB(){
6    LOCK(mutex);
7     if (order==1) {
8       order=2;
9       send_data(a[]);}
10   UNLOCK(mutex);
11   LOCK(mutex);
12    if (order==0) {
13      order=1;
14      a[]=get_data();}
15   UNLOCK(mutex);
16 }
```

(a) Code snippet **(b)** Program Dependency

Fig. 6: Control and data dependency analysis.

in Fig. 5(b), ($m1$) is injected into two *CPE* functions, and ($m2$ and $m3$) is only in one *CPE* function. When these two *CPE* functions are invoked in parallel, three *malware sections* would be executed with different orders, which may face similar faults as *SPE*: *Repeated execution* and *Execution with wrong order*. Thus, the control module is also needed to ensure the *malware components* are executed correctly.

Both *CPE* and *SPE* functions could be selected to inject *malware sections*. We need to analyze at least two different *CPE* functions, but only one *SPE* function. Meanwhile, the activation methods of malware injected into *CPE* and *SPE* are different. Thus, we focus on the *SPE* function-targeted CLB technique in this paper due to the page limit.

B. Stealthiness of CLB

Since the CLB technique partitions the malware and hides its components into various places in a hosting program, it is almost impossible for static analysis techniques to detect malware, as confirmed by the experiments in Sec. V-B. In this section, we will show that the CLB technique could also evade current dynamic analysis techniques.

Dynamic malware detection exploits control dependency and data dependency to find suspicious executions path, and guides dynamic analysis to identify malicious behavior. Such an approach is not applicable to defend CLB. As shown in Fig. 6(a), there are two malware sections. An array *a[]* is used to store the sensitive data between the two sections. This malware is detected if *send_data* is executed after *get_data* in one execution. As shown in Fig. 6(b), we extract the control dependency of the malware sample in red. It shows that *send_data* at Line 9 depends on Line 7 and *get_data* at Line 14 depends on Line 12. The data dependency of the two global variables *order* and *a[]* is depicted in blue. For *order*, the conditional statement at Line 12 is dependent on the assignment statement at Line 8. For *a[]*, note that the *a[]* transmitted by *send_data* (Line 9) is irrelevant to the *a[]* obtained by *get_data* (Line 14). Considering the control dependency and data dependency, there is no execution path containing the *get-send* pattern. Therefore, the dependencies could not guide dynamic analysis to defend CLB. Moreover, we envision many stealth techniques can be implemented with the CLB technique to make it even harder for current malware analysis. For example, side-channel leakage can be used to provide a more stealthy data flow among various malware sections [6]–[8].

Another attempted defense against CLB is to explore all the possible executions, by integrating dynamic malware detection techniques with concurrent testing, such as model checking (e.g. ESBMC [9]) or symbolic execution (e.g. DTAM [10], Proactive-Debugger [11], Conc-iSE [12]). These tools can be applied to explore all interleavings. However, this approach is not practical and scalable due to the inherent issues of model checking and symbolic execution, such as state explosion, availability of source code and nonlinear computation, the sheer size of interleavings.

IV. Controllable Probabilistic Activation

A. Definition

The malicious behavior in a sequential program is triggered when the input vector (*in*) is among the activation inputs IN_{ACT}. In most cases, the execution of a sequential program is deterministic. Then, the activation of sequential malware can be formally presented as

$$P(malware_{seq} = active | in \in IN_{ACT}) = 1 \tag{1}$$

where $P(\cdot)$ is the probability function.

As demonstrated by Fig. 3, the triggering condition in Eq. (1) cannot guarantee the activation of the malicious behavior in a concurrent program because the execution traces can be different with the same input. Thus Eq. (1), which is valid for sequential programs, is no longer valid for concurrent programs. We define *probabilistic activation* for concurrent malware as

$$P(malware_{con} = active | in \in IN_{ACT}) = \theta \tag{2}$$

Eq. (2) states that a concurrent malware is triggered with a probability of $\theta \in [0, 1]$ when its input is among the activation inputs. A lower θ indicates that a concurrent malware sample is more stealthy and less likely to be triggered under a normal execution environment.

However, θ alone does not reveal the severity as it does not indicate how likely the concurrent malware can be triggered by an attacker. Thus, we introduce the concept of *controllable probabilistic activation* (CPA) as below.

$$\begin{aligned} &P(ConcS = active | in \in IN_{ACT}) = \theta \\ &P(ConcS = active | in \in IN_{ACT} \wedge side\_cond) = \gamma \end{aligned} \tag{3}$$

where $side\_cond$ is a side condition that is irrelevant to inputs but can be controlled or influenced by an attacker. With a side condition, the probability of activating a ConcSpectre malware sample γ can be significantly greater than θ. Therefore, θ and γ represent the *stealthiness* and *controllability*, respectively. The gap $\delta = \gamma - \theta$ can indicate the severity of a piece of concurrent malware.

B. Probabilistic Activation

For each thread interleaving, there is an execution trace that contains malware sections. The number of execution traces with different ordering of malware sections is $(a * b)!/(b!)^a$, where a and b are the number of threads and malware sections, respectively. The activation of ConcSpectre malware relies on whether all malware sections have been executed successfully in the intended order. We can calculate the rate of malware-activated traces by traversing all possible thread interleavings. In a real system, the occurrence probabilities of thread interleaving are affected by the predetermined malware-activation order, and various uncertain factors, such as synchronization primitives in host programs, OS scheduling mechanism, system load, hardware, etc. The rate of activation order is considered as a reference for activation strategy selection.

Consider the ConcSpectre malware sample in Fig. 6(a), where two threads execute two malware sections, respectively. There are $(2 * 2)!/(2!)^2 = 6$ possible thread interleavings, as shown in the first column of Table I. The activation strategy of the malware sample is that *section 1 should be executed after the execution of section 2* ($S_2 < S_1$). The thread interleavings, execution traces and the activation states are shown in Columns 2, 3 and 4, respectively. Malicious behavior would be activated in two traces. Assuming the activation strategy is revised to *section 1 should be executed before section 2* ($S_1 < S_2$), malicious behavior is then activated in all traces. This is illustrated in the last two columns in Table I. The reason is that malware section 1 is always executed before malware section 2 in any individual thread.

We define the order of two adjacent malware sections as ($S_i < S_j$). If $i < j$, then ($S_i < S_j$) is an *Ordered Pair*. On the other hand, if $i > j$, ($S_i < S_j$) is a *Reverse-Order Pair*, and $i - j$ is the *Reverse-Order Degree*. Since all *Ordered Pairs* are satisfied in any individual thread, the *Reverse-Order Pair* in the activation strategy is the key to decide whether the malicious behavior can be triggered.

In our work, we have analyzed nine situations, including 2–4 threads and 2–4 *malware sections* per thread. The number of malware-activated thread interleavings is shown in Table II. With activation strategy "$S_1 < S_2 < S_3$", malicious behavior would always be triggered. However, with "$S_3 < S_2 < S_1$", only 0%, 6% and 14% thread interleavings would trigger the malicious behavior when there are 2–4 threads, respectively.

Observation 1: *With more reversed orders and a higher reverse-order degree in an activation strategy, fewer thread interleavings can activate a ConcSpectre malware sample.*

C. Load-based Controllable Probabilistic Activation on Windows

According to Eq. (3), an exploitable piece of ConcSpectre malware requires a side condition to improve the activation probability. A feasible side condition must meet two requirements: (1) accessible, and (2) irrelevant to the malware itself.

On Windows OS, the scheduler divides the available processor time in a round-robin fashion among the processes or threads following scheduling priority. Thus, there are three

TABLE I: Thread interleavings and their activation states.

ID	Interleaving	Activation Strategy: $S_2 < S_1$		Activation Strategy: $S_1 < S_2$	
		Execution trace	Result	Execution trace	Result
$\pi1$	T1-T1-T2-T2	T1(1,F)-T1(2,T)-T2(1,T)-T2(2,F)	Active	T1(1,T)-T1(2,T)-T2(1,F)-T2(2,F)	Active
$\pi2$	T1-T2-T1-T2	T1(1,F)-T2(1,F)-T1(2,T)-T2(2,F)	Inactive	T1(1,T)-T2(1,F)-T1(2,T)-T2(2,F)	Active
$\pi3$	T1-T2-T2-T1	T1(1,F)-T2(1,F)-T2(2,T)-T1(2,F)	Inactive	T1(1,T)-T2(2,F)-T2(1,T)-T1(2,F)	Active
$\pi4$	T2-T2-T1-T1	T2(1,F)-T2(2,T)-T1(1,T)-T1(2,F)	Active	T2(1,T)-T2(2,T)-T1(1,F)-T1(2,F)	Active
$\pi5$	T2-T1-T1-T2	T2(1,F)-T1(1,F)-T1(2,T)-T2(2,F)	Inactive	T2(1,T)-T1(1,F)-T1(2,T)-T2(2,F)	Active
$\pi6$	T2-T1-T2-T1	T2(1,F)-T1(1,F)-T2(2,T)-T1(2,F)	Inactive	T2(1,T)-T1(1,F)-T2(2,T)-T1(2,F)	Active

TABLE II: Number of effective interleavings (three *malware sections*) under various activation strategies.

#Thread	#Section	#Interleavings	Activation Strategy					
			$S_1 < S_2 < S_3$	$S_1 < S_3 < S_2$	$S_2 < S_1 < S_3$	$S_2 < S_3 < S_1$	$S_3 < S_1 < S_2$	$S_3 < S_2 < S_1$
2	3	20	20	8	8	2	2	0
3	3	1,680	1,680	1,140	1,140	384	384	96
4	3	396,000	396,000	309,120	309,120	132,000	132,000	56,832

variable factors to influence thread scheduling: *available processor time*, *scheduling priority* and *round-robin mechanism*. Obviously, it is difficult to access and control the *scheduling priority* or *round-robin* on the victim's system. In our work, we find the *available processor time* is relevant to system load that can be influenced remotely.

Assume that the example in Fig. 6 runs on a dual-core CPU system. When the system is in an idle state, the scheduler may assign CPU1 and CPU2 to two threads. Two threads can start to execute *malware section* 1 simultaneously, as shown in Fig. 7(a). Since there is LOCK to maintain the atomicity of all *malware sections*, two threads would be executed one by one (as the execution trace $\pi2$ in Table I). In such cases, the malware would be triggered with low probability, since the second *malware section* is unlikely to be activated.

When the system is in a high load state (e.g., CPU 2 is occupied by a high priority task), only one thread can obtain the resource to execute. Thus, two threads would be executed sequentially, as shown in Figures 7(b). Two *malware sections* would be activated within two threads, and the ConcSpectre malware would be triggered with high probability.

When the threads are executed concurrently, the *malware sections* in different threads may start in the same time slice. There will be fewer reverse orders in an execution trace. When the threads are executed sequentially, there will be more reverse orders. Thus, we could disturb the thread scheduling on victim's system by influencing its load. In particular, we can increase the occurrence probability of thread interleavings with more reverse orders by increasing its workload, which leads to the *Load-based CPA*.

To verify the *Load-based CPA*, we run a concurrent program on a Windows server with Intel Xeon CPU E7-4850 and 8GB memory. We create four threads to execute four functions (S_1 to S_4) that read and write some local files with the same order. We simulate nine groups of experiments. In each group, the concurrent program runs 10,000 times, and we execute 0 to 8 programs with infinite-loops to simulate the system load from 0% to 100%. All execution traces of the concurrent program are recorded. Then, we match all possible control strategies in all traces to calculate their activation probabilities. As shown in Fig. 8, the activation probability increases significantly when

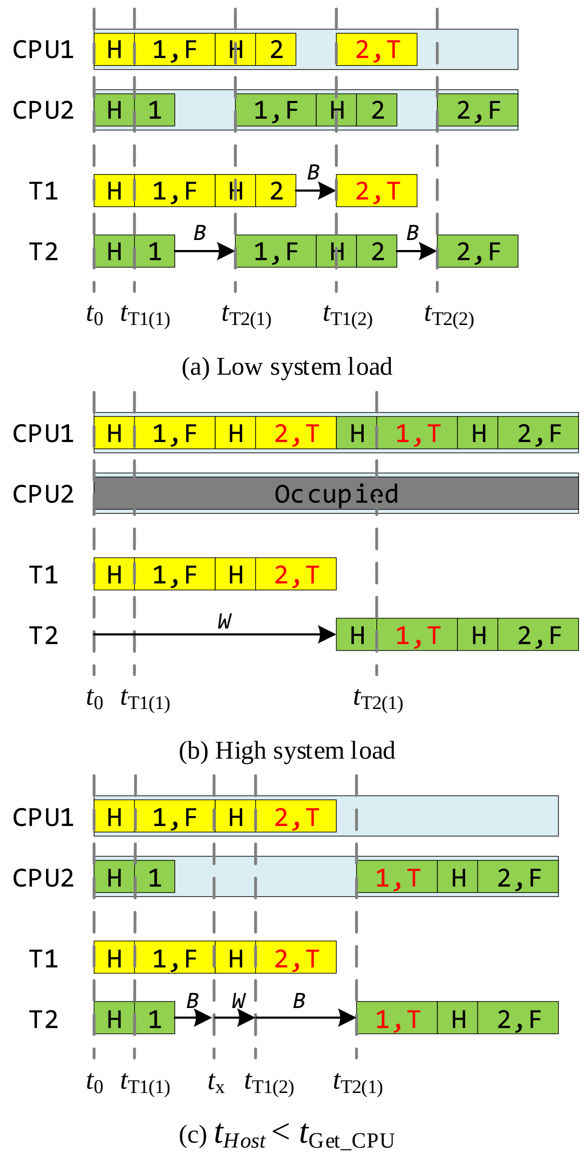

CPU1
CPU2
T1
T2
H
1,F
2,T
1,T
2,F
B
W
Occupied
t_0 $t_{T1(1)}$ $t_{T2(1)}$ $t_{T1(2)}$ $t_{T2(2)}$ t_x
(a) Low system load
(b) High system load
(c) $t_{Host} < t_{Get\_CPU}$

Fig. 7: Thread interleavings (1,T/1,F means the malware section 1 has (not) been executed, H is the execution of the host program, B means the thread is blocked by the LOCK, and W means the thread is waiting for processor slicing).

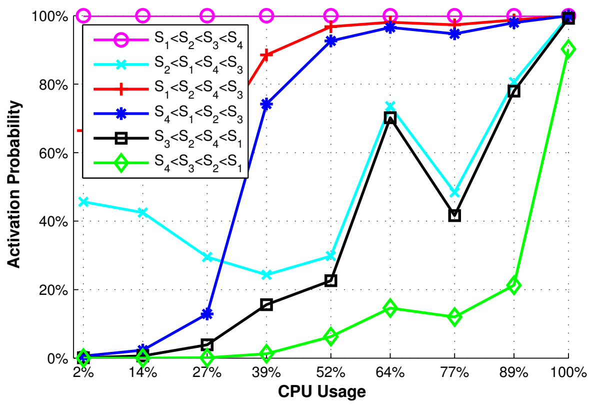


Fig. 8: Activation probability under different activation strategies.

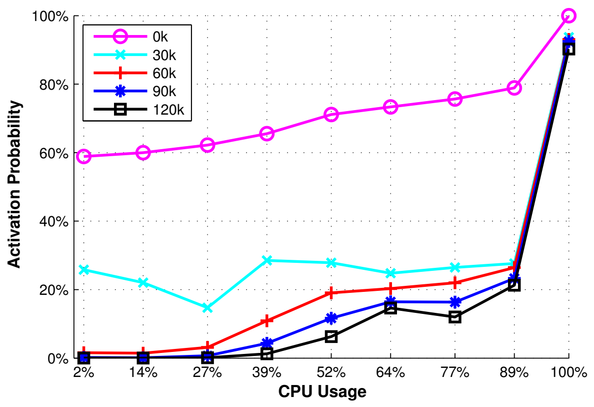


Fig. 9: Activation probability of $S_4 < S_3 < S_2 < S_1$.

the CPU usage rises. In particular, the occurrence rate of the full reverse-order sequence $S_4 < S_3 < S_2 < S_1$ is dramatically increased from 0.04% (2% CPU usage, i.e., the average system load) to 90.24% (100% CPU usage). Note that the sequence $S_1 < S_2 < S_3 < S_4$ would be activated within each individual thread, so its activation probability is always 100%.

Observation 2: *We can significantly change the activation probability of reverse-order control strategy by influencing the workload on the victim's system.*

The running time of *malware sections* is an important factor in deciding when and on which thread they execute. On Windows OS, the scheduler allocates a processor *time slice* (approximately 20 milliseconds) for each thread it executes. The running thread is suspended when its *time slice* elapses, allowing another thread to run [13]. Thus, if the interval between two *malware sections* is too short, the expected thread interleaving may be changed. As shown in Fig. 7(c), Thread 2 will be blocked when *malware section* 1 has been locked by Thread 1 at $t_{T1(1)}$. At t_x, Thread 1 releases the LOCK and Thread 2 starts to request the processor again. If Thread 1 starts to execute *malware section* 2 before Thread 2 gets the processor slicing, Thread 2 will be blocked again. The expected thread switching in Fig. 7(a) would not happen. And, all *malware sections* would be executed successfully with high probability, regardless of the system loads.

In our experiments, we design a concurrent program, in which four threads execute four functions in the same order. These functions only record their execution time. We run the program 10,000 times under different system loads. As shown in Fig. 9 (the pink line marked as 0k), more than 6,000 executions contain the sequence $S_4 < S_3 < S_2 < S_1$ when CPU usage is less than 40%, which are much higher than the experiments in Fig. 8.

Thus, we add a waiting section to extend the running time of *malware sections*. As shown in Fig. 9, four groups of empty loops are added into the *malware sections*, in which the number of loops is 30k, 60k, 90k and 120k. We run the concurrent program 10,000 times under different system loads. It presents expected activation probability during low and high system load. If a 90k-empty-loop is added, there are fewer than 0.68% executions containing the sequence $S_4 < S_3 < S_2 < S_1$ when CPU usage is less than 30%; but as high as 92% when CPU usage is 100%.

Observation 3: *To obtain different activation probabilities under different workloads, the running time of malware sections should be longer than the time slice of thread scheduling.*

V. Evaluation

In this section, we select four widely-studied open-source malware samples from VX Heavens [14], including *Bull-Moose*, *Branko*, *Hunatcha* and *Hunatchab*, as shown in Table III [15]. *BullMoose* is selected as the demo sample to demonstrate how to generate and remotely trigger ConcSpectre malware. All four malware are hidden into benign concurrent programs to construct ConcSpectre and verify its stealth against current malware analysis techniques.

As shown in Table IV, we select nine programs from the well-known concurrent testing benchmark SPLASH [16] and one program from Microsoft Open Source Code [17] as the host programs. By analyzing their source codes, we identify 217 SPE functions from these programs. Then, we select 1 to 15 SPE functions in each host program that result in 77 suitable injection points in total, which could be invoked by at least four threads in parallel.

In this paper, we generate over 1,000 samples of Conc-Spectre malware by injecting four malware samples and their variants into ten benign concurrent programs with different activation strategies at the code level. The processes of malware partition, benign program analysis and ConcSpectre construction are at code level. By debugging and compiling these samples, we generate the executables of all original malware and ConcSpectre samples, and pass them to Virus-Total (integrating 67 anti-virus engines) and four dynamic malware analysis systems to demonstrate: (1) How to generate real ConcSpectre malware samples; (2) How to trigger Conc-Spectre malware remotely; (3) Whether anti-virus systems can detect ConcSpectre malware; (4) Whether dynamic malware analysis systems can detect ConcSpectre malware.

TABLE III: Malware samples (LOC is the lines of code).

Malware	Type	LOC
BullMoose	Trojan	30
Branko	Worm	266
Hunatcha	Worm	164
Hunatchab	Worm	339

TABLE IV: Concurrent programs (LOC is the lines of code, #Function, #SPE and #Injection Point are the number of functions, SPE functions and injection points, respectively).

Program	LOC	#Function	#SPE	#Injection Point
cholesky	5491	127	102	5
fft	1482	20	14	8
lu_c	1401	19	15	6
lu_nc	1182	19	15	6
ocean_c	5408	21	19	15
ocean_nc	3561	21	19	15
radix	1547	12	8	7
water_n	2593	16	12	7
water_s	3139	16	12	7
Multiverso	16254	991	1	1

A. ConcSpectre Malware Generation

Fig. 10 is the workflow of ConcSpectre malware generation. All ConcSpectre malware samples are constructed with C/C++ (mingw32-gcc 6.3.0) and Pthread (mingw32-libpthreadgc, version: 2.10-pre-20160821-1) on Windows 7 and Windows 10. In this section, we illustrate the detailed steps with a real case:

- malware=*BullMoose*, host program=*fft*;
- CPA is set as the system load-based strategy. The activation probability should be lower than 5% during normal system load (CPU usage is less than 25%), and higher than 50% during high load (CPU usage is higher than 75%).
- A global variable is used to control the execution of each *malware section*;
- *Self-parallel-execution* (SPE) functions in the host program are chosen to inject the malicious code.

In the *Malware Separation* module, a static analysis technique is applied to extract the control dependency and data dependency of malware. These dependencies are used to guide malware partition to make sure the relation between different malware fragments is as little as possible. Then, each component is checked with various anti-malware systems. If there are any abnormal alarms, we need to partition the abnormal component again or apply obfuscation and shelling techniques to make sure that it does not cause any alarms. By analyzing the source code of *BullMoose*, we partition it into four components to ensure each component would not be classified as a malicious program. A global variable is added to control the execution order of the malware components[1].

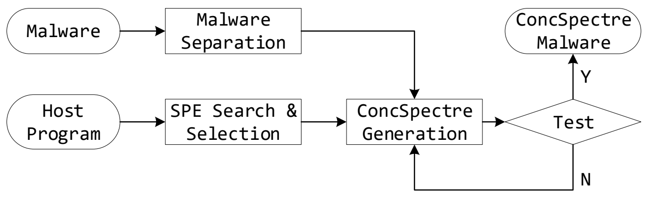


Fig. 10: Workflow of ConcSpectre generation.

[1]The code of four malware sections is provided on https://git.io/CLB.

In the *SPE Search & Selection* module, we search the source code of the host program to find all possible self-parallel-execution functions. Meanwhile, we also execute the host program and monitor its execution traces to check how many threads are created. In *fft*, 14 SPE functions are found, in which eight functions could invoke at least four threads in parallel. In the demo case, we select function "Slavestart" as the injection point.

In the *ConcSpectre Generation* module, all *malware sections* would be embedded into host programs at selected points. To ensure the integrity of malware's data flow, we identify the shared variables of different *malware sections* and set them as global variables. Other possible errors, such as variable renaming, read-write collision, etc., could be fixed during compiling. Then, the executables of the ConcSpectre malware could be generated.

The activation probability of ConcSpectre malware relies on various factors, including OS, thread scheduler, host program, malware, etc. In our work, we firstly select a control strategy guided by the rate of malware-activated trace (e.g., Tables II) and generate a ConcSpectre malware sample. Then, the *Test* module is applied to execute the sample under different system loads to calculate its activation probability. If the probabilities meet the requirement, we get a satisfactory sample; if not, we generate a new sample with another activation strategy.

In our experiments, we have simulated various sequences under different system loads with empty loops, and recorded their activation probabilities. According to the requirement in current case, we select the first variant whose activation order is $S_4 < S_1 < S_2 < S_3$ and set the number of empty loop as 120k to generate a ConcSpectre malware sample, named as *BullMoose-fft-C*1.

In this paper, we introduce how to construct ConcSpectre malware with *SPE* function and under specific input. In most concurrent programs, there are more *CPE* functions than *SPE*. The attackers can use a similar method to partition malware and inject their components into *CPE* functions to bypass the static malware detection. However, the activation strategy would be more complicated, since *CPE* functions would present different privileges, external conditions, synchronization constraints, etc. The input of host program would also affect the thread interleavings when concurrent programs execute. We randomly select the input and monitor the execution to search suitable injection points in this paper. Advanced techniques, such as Logic Bomb, could be exploited to select activation input of ConcSpectre.

B. Remote Triggering ConcSpectre Malware

We further demonstrate the feasibility of remote controlling to activate *BullMoose-fft-C*1 in a real network. We set up a victim system (with Intel CORE I5 3470, 16G, JAVA 8) with Red5 Media Server [18] as a local web server. *BullMoose-fft-C*1 is injected into one web page. Initially, we use a script file to request this page for 100 times. However, *BullMoose-fft-C*1 is never triggered. This is actually reasonable as the CPU usage is only up to 3% and there is only one user.

To simulate the real case with large-scale concurrent accesses, we run a real-world HTTP DDoS test tool GoldenEye [19] to visit our web server, and configure it to work under five different groups of users: 3, 6, 10, 20, and 30. Each user visits the web server with ten concurrent sockets. Note that these sockets do not invoke the web page containing ConcSpectre malware. At the same time, we also launch our script to visit the web page that contains *BullMoose-fft-C*1 for 100 times. As shown in Fig. 11, we see that *BullMoose-fft-C*1 is activated with different probabilities. In particular, when the number of attackers increases from 0 to 3 and 30, the average CPU usage on victim's system increases from 3% to 23% and 70%, respectively. Consequently, the activation probability of *BullMoose-fft-C*1 increases from 0% to 4% and 51%, respectively.

Although the probability of activation in real-world experiments (Fig. 11) differs from the simulation (Fig. 8), these real-world experiments demonstrate that ConcSpectre malware can be remotely triggered with high probability if attackers can perturb the system load.

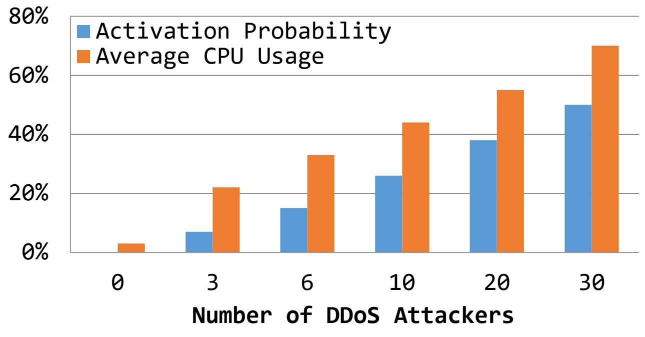


Fig. 11: CPU usage and activation probability.

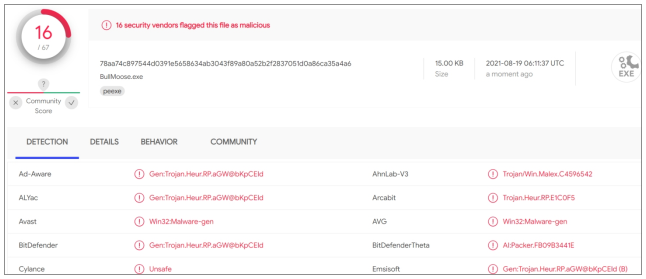
C. *ConcSpectre vs Anti-virus System*

One design requirement of ConcSpectre malware is to escape from the anti-virus systems. Therefore, we select all anti-virus engines from VirusTotal to analyze generated ConcSpectre malware. Note that these engines are well-configured and also up-to-date. The set includes widely used ones such as McAfee, Microsoft and Symantec.

We submit the executable file of *BullMoose* and *BullMoose-fft-C*1 to VirusTotal. As shown in Fig. 12, the results are impressive: (1) the original *Bullmoose* is detected by 16 out of 67 engines in Fig. 12(a); (2) Only two engines ("Cylance" and "MaxSecure") could detect threat from the ConcSpectre samples as in Fig. 12(b). After further analysis, these two alarms are both false postives, which are inherited from the host program *fft*.

To justify whether the ConcSpectre can bypass various anti-virus engines, we generate 924 samples by injecting four malware into 77 injection points of all host programs with the following three activation strategies: C2 ($S4 < S3 < S2 < S1$), C3 ($S4 < S1 < S3 < S2$), and C4 ($S4 < S2 < S3 < S1$). We submit four original malware samples and 924 generated ConcSpectre samples to VirusTotal, the results are similar to the previous one. The original malware samples could be detected by most of the engines. Meanwhile, few engines could detect ConcSpectre samples correctly.

From the result, even the widely-used and up-to-date anti-virus engines fail to identify true threats of ConcSpectre malware. This exposes potential threats of ConcSpectre.

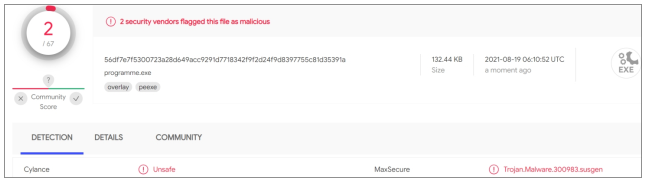


16 security vendors flagged this file as malicious

78aa74c897544d0391e5658634ab3043f89a80a52b2f2837051d0a86ca35a4a6
BullMoose.exe
peexe
15.00 KB Size
2021-08-19 06:11:37 UTC a moment ago

Ad-Aware	Gen:Trojan.Heur.RP.aGW@bKpCEId	AhnLab-V3	Trojan/Win.Malex.C4596542
ALYac	Gen:Trojan.Heur.RP.aGW@bKpCEId	Arcabit	Trojan.Heur.RP.E1C0F5
Avast	Win32:Malware-gen	AVG	Win32:Malware-gen
BitDefender	Gen:Trojan.Heur.RP.aGW@bKpCEId	BitDefenderTheta	AI:Packer.FB09B3441E
Cylance	Unsafe	Emsisoft	Gen:Trojan.Heur.RP.aGW@bKpCEId (B)

(a) BullMoose



2 security vendors flagged this file as malicious

56df7e7f5300723a28d649acc9291d7718342f9f2d24f9d8397755c81d35391a
programme.exe
overlay peexe
132.44 KB Size
2021-08-19 06:10:52 UTC a moment ago

Cylance	Unsafe	MaxSecure	Trojan.Malware.300983.susgen

(b) BullMoose-fft-C1

Fig. 12: Detection results of VirusTotal.

D. *ConcSpectre vs Dynamic Malware Analysis*

To monitor the dynamic execution of a program and report any suspicious operations, we select four popular dynamic malware analysis systems, including: *Jevereg* [20], Falcon [21], Anlyz [22] and ANYRUN [23]. They are developed on different well-known sandboxs [10], [24], [25], such as Amnpardaz, Falcon, etc.

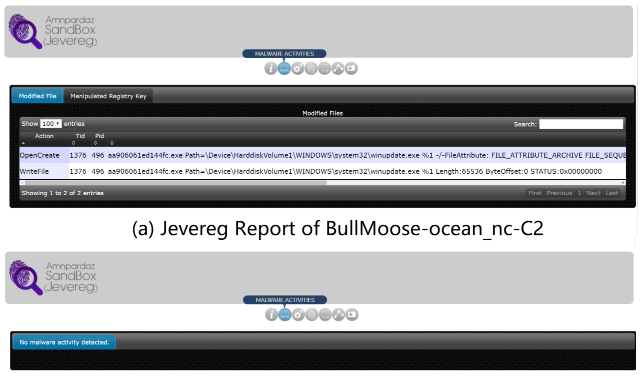
Initially, we select the ConcSpectre samples generated from the *BullMoose* in Table V. Since the numbers of SPE functions in different host programs are different, the number of malware samples in each host program is also different (listed as #Sample). One special variant is generated with the activation strategy $S1 < S2 < S3 < S4$, which would be always be activated. We inject this variant into host program *fft* to generate the ConcSpectre malware sample *BullMoose-fft-A*.

TABLE V: Detection results of dynamic malware analysis (three different activation strategies C2, C3 and C4 are marked as -C, the strategy $S1 < S2 < S3 < S4$ is marked as -A, #Sample is the number of malware samples).

Malware (#Sample)	Jevereg	Falcon	Anlyz	ANYRUN
BullMoose (1)	1	1	1	1
BullMoose-fft-A (1)	1	1	1	1
BullMoose-cholesky-C (15)	0	0	0	0
BullMoose-fft-C (24)	18	0	0	0
BullMoose-lu_c-C (18)	16	0	0	6
BullMoose-lu_nc-C (18)	17	0	0	6
BullMoose-ocean_c-C (45)	41	0	0	30
BullMoose-ocean_nc-C (45)	41	0	0	33
BullMoose-radix-C (21)	17	0	0	12
BullMoose-water_n-C (21)	0	0	0	0
BullMoose-water_s-C (21)	0	0	0	0
BullMoose-Multiverso-C (3)	0	0	0	0

Together with *Bullmoose*, we submit 233 programs to four dynamic malware analysis systems. As shown in Table V, *Bullmoose* and *BullMoose-fft-A* have been detected by all systems. For the ConcSpectre samples, the results are different. In particular, 150 samples from 6 host programs have been identified as suspicious programs by *Jevereg*, 87 samples have been detected by *ANYRUN*, and all samples have been identified as benign by the rest two systems. As shown in Fig. 13(a), *Jevereg* detects two suspicious operations (OpenCreate and WriterFile) on the files in /System32.

One possible reason for the high detection rate is that the CPU resource is limited on the server of *Jevereg* and *ANYRUN*. For example, when the server allocates one thread to execute the uploaded samples, they would be executed sequentially, and the malware would be triggered with high probability as in Sec. IV-C. If our conjecture is true, *Jevereg* and *ANYRUN* would fail on detect the ConcSpectre whose activation condition contains parallel sequence. Thus, we design a hybrid activation sequence: C5 ($S1 < S1 < S1 < S4 < S3 < S2$) to generate the fourth variant of *Bullmoose*, in which $S1$ would be executed three times before $S4$. The first part $S1 < S1 < S1$ would be activated with high probability under low CPU load, and the second part $S4 < S3 < S2$ would be satisfied under high CPU load. Thus, it is not easy to trigger this sequence under any conditions of CPU load. We generate a new group of ConcSpectre malware samples containing this variant. From Table VI, no samples are detected as malware under the new activation strategy. Compared to its high detection probability on the previous strategy, *Jevereg* and *ANYRUN* fail on the parallel sequence. It proves our conjecture that the *Jevereg* and *ANYRUN* servers allocate limited resources for each application.



Show 100 entries Search:

Action	Tid	Pid	
OpenCreate	1376	496	aa906061ed144fc.exe Path=\Device\HarddiskVolume1\WINDOWS\system32\winupdate.exe %1 -/-FileAttribute: FILE_ATTRIBUTE_ARCHIVE FILE_SEQUE
WriteFile	1376	496	aa906061ed144fc.exe Path=\Device\HarddiskVolume1\WINDOWS\system32\winupdate.exe %1 Length:65536 ByteOffset:0 STATUS:0x00000000

Showing 1 to 2 of 2 entries First Previous 1 Next Last

(a) Jevereg Report of BullMoose-ocean_nc-C2

No malware activity detected.

(b) Jevereg Report of BullMoose-ocean_nc-C5

Fig. 13: Detection results of Jevereg.

These two sets of experiments show that ConcSpectre malware can bypass many current malware detection systems. Moreover, they can also escape from being detected under high system load by introducing the parallel-execution requirement into the activation sequence.

We further inject the rest three malware samples into all host programs using the hybrid activation strategy C5. Since the number restrictions on sample uploading of these online analysis systems, we randomly select one ConcSpectre sample for each host program and malware, in total 30. Together with

TABLE VI: Detection results of malware with hybrid activation strategy.

Malware	Jevereg	ANYRUN
BullMoose-cholesky-C5	0	0
BullMoose-fft-C5	0	0
BullMoose-lu_c-C5	0	0
BullMoose-lu_nc-C5	0	0
BullMoose-ocean_c-C5	0	0
BullMoose-ocean_nc-C5	0	0
BullMoose-radix-C5	0	0
BullMoose-water_n-C5	0	0
BullMoose-water_s-C5	0	0
BullMoose-Multiverso-C5	0	0

TABLE VII: Detection results of other malware samples (#Sample is the number of malware samples).

Malware (#Sample)	Jevereg	Falcon	Anlyz	ANYRUN
Branko (1)	1	1	1	1
Branko-C5 (10)	0	0	0	0
Hunatcha (1)	1	1	1	1
Hunatcha-C5 (10)	0	0	0	0
Hunatchab (1)	1	1	1	1
Hunatchab-C5 (10)	0	0	0	0

three original malware samples, 33 samples are submitted to four analysis systems. As shown in Table VII, all original malware samples would be detected as malware by all systems. Meanwhile, no ConcSpectre samples have been detected.

VI. Defense and Discussion

It is intractable for analysts to detect and defend ConcSpectre malware, which might be widely used in multi-core systems and multi-threaded programs. In this section, we present and discuss several possible solutions for detection and defense.

Exhaustive Examination. It is well known that model checking is a powerful technique for exploring the whole state space for a program [26]–[28]. We can implement the ConcSpectre detector based on the model checker, by adding the activation property of malicious behaviors on the basis of current model checkers [9], [29], [30]. We apply the latest version (2021) of ESBMC [9] to analyze *cholesky* under a fixed input. The verification of one input could not finish within 3600s. Since the scalability of model checking is limited by state space explosion, it could only be applied for small programs.

In addition, the symbolic execution is also used to analyze the complex behavior among different threads [31]–[33]. Its advantage lies in the capability of automatically finding intricate interleavings. It encodes an execution trace of a multi-threaded program and the activation condition of malicious behavior into a symbolic formula and then symbolically seeks an objective interleaving by solving the formula with an SMT solver [34]. We reproduce the encoding method in [35] and use it to verify an assertion in *cholesky*. For an execution trace with 250k events, it spends 35s on looking for an interleaving triggering the assertion. However, in real-world systems, their executions are extremely long. Meanwhile, the activation condition of malicious behavior is more complicated than an assertion. In short, scalability is the major problem for *exhaustive examination* approaches.

Affecting thread scheduling. Various methods of affecting scheduler are applied in concurrent program testing to cover as many different interleavings as possible [36]–[38]. A common method is to randomly insert sleep statements or empty loops into programs and keep running the programs up to a time bound. It would increase the probability of exposing the malicious code. However, there are many questions: 1) how to select the place to inject these extra delays. It is another state explosion of various threads and parallel functions; and 2) how to set the time bound. There is a trade-off between interleaving coverage and testing cost.

Another method is to randomly change the CPU usages during analyzing software, which is the same as the CPA technique we proposed. As shown in Sec. V-D, the attackers can design various CPA strategies which would not be easily activated under both low and high system load. Thus, the defender can randomly assign the CPU resource during testing. It would increase the chance to detect the hidden malware.

Serialization. The simultaneous execution of many threads is the fundamental cause of ConcSpectre malware being activated while bypassing the detection. A compromise approach is to serialize the temporal orderings of shared access points by inserting synchronization statements into program code. Thus, we only permit the executions of benign temporal orderings. It is an effective defense method, which simultaneously brings performance reduction for multi-task processing.

In summary, it is possible to disturb the CPA strategies and prevent the ConcSpectre malware from being triggered. For example, the defender can change the thread scheduling by randomly injecting sleep statements, keeping the workload stable with various load balance techniques, and serializing parts of the program. Although these techniques cannot guarantee to detect the ConcSpectre malware or prevent its activation, they can markedly alleviate the impact of ConcSpectre malware.

VII. Related Work

Current stealth malware techniques could be classified into four types: rootkit, code mutation, anti-emulation and targeting mechanism [39]. Code mutation aims to alter the appearance of malicious code to bypass malware detection systems based on pattern-matching algorithms. The malicious code would be triggered under specific inputs or conditions. ConcSpectre applies similar methods to partition malware and injects them into benign programs. However, to the best of our knowledge, it is the first malware technique to hide its malicious behavior by exploiting the non-deterministic thread interleavings, which can bypass current static and dynamic malware detection even when the activation input is known.

There are some similar works, such as multi-stage malware [40], concurrency attack [41], [42] and cooperative attack [43]–[45] . Some well-known computer viruses, such as Internet worm [46] and RMNS [47], are multi-stage, which achieve an attack by cooperating distributed tasks across multiple processes. Xu et al. find the Inter-Component Communication (ICC) mechanism can be exploited by malware to obfuscate malicious behaviors and bypass existing detection methods. Thus, they design an ICC-Based malware detector to detect the malicious behaviors hidden in different components [48]. A cooperative attack is trickier in concealing malware. It spatially divides and places the system-call sequences of malware into separate processes. There is no single process that performs the malicious actions, so any attempt to monitor individual processes for malicious behaviors would fail. Meanwhile, these processes can cooperate to perform malicious actions in a temporal order. Especially, Wang et al. design an attacker that can bypass the Apple Review, remotely exploit the planted vulnerabilities and assemble the malicious logic at runtime by chaining the code gadgets together [45]. However, the cooperative attack suffers from two limitations: 1) it is difficult for such malware to be applied in a real-time attack [49], and 2) any failure in any process will lead to the failure of the entire process [43]. This spatial and temporal division of the malicious behaviors is similar to ConcSpectre. The major difference is that ConcSpectre is implemented on different threads, without cooperating with other hosts or processors. Thus, ConcSpectre would be easier to control for attackers.

Yang et al. discover that errors in concurrent programs can lead to concurrency attacks, and firstly prove that concurrency attacks are indeed viable [41], [42]. The major difference between ConcSpectre and concurrency attacks is that ConcSpectre attackers hide the malware within concurrency programs, while concurrency attackers exploit the concurrency bugs. As discussed in the previous section, it is difficult for an analyst to trigger and detect the malicious code in ConcSpectre malware.

VIII. Conclusion

In this paper, we present ConcSpectre, a type of malware that exploits concurrency. Its implementation can be based on CLB, a new stealth malware technique, and CPA, a new malware activation technique. More than 1,000 ConcSpectre malware samples have been generated based on four real malicious programs and ten concurrent programs. All of them can bypass most of the anti-virus engines in VirusTotal and four online dynamic malware detection systems. Experiment results reveal the threat of ConcSpectre, which calls for an urgent redesign of malware detection techniques for concurrent programs.

We have explored several software testing techniques to detect ConcSpectre malware. However, scalability and correctness are two major issues of current concurrent software testing tools. A tentative solution is to actively control thread scheduling or even serialize concurrent programs so that thread scheduling cannot be perturbed. However, such a strategy would incur a significant burden on programmers and also limit concurrency.

In this paper, we have discussed the *SPE* function-targeted CLB and CPA techniques. *CPE* is also common in concurrent programs. We can use similar methods to partition malware and inject their components into *CPE* functions to bypass static malware detection. However, the activation strategy would be more complicated, since the *CPE* functions would present different privileges, external conditions, synchronization constraints, etc. In the future, we plan to further investigate the *CPE* function-targeted ConcSpectre techniques.

References

[1] M. Lipp, M. Schwarz, D. Gruss, T. Prescher, W. Haas, S. Mangard, P. Kocher, D. Genkin, Y. Yarom, and M. Hamburg, "Meltdown," https://arxiv.org/abs/1801.01207, 2017.

[2] P. Kocher, J. Horn, A. Fogh, D. Genkin, D. Gruss, W. Haas, M. Hamburg, M. Lipp, S. Mangard, T. Prescher, M. Schwarz, and Y. Yarom, "Spectre attacks: Exploiting speculative execution," in *2019 IEEE Symposium on Security and Privacy (SP)*, 2019, pp. 1–19.

[3] M. Musuvathi and S. Qadeer, "Iterative context bounding for systematic testing of multithreaded programs," *ACM Sigplan Notices*, vol. 42, no. 6, pp. 446–455, 2007.

[4] G. Suarez-Tangil, J. E. Tapiador, P. Peris-Lopez, and A. Ribagorda, "Evolution, detection and analysis of malware for smart devices," *IEEE Communications Surveys & Tutorials*, vol. 16, no. 2, pp. 961–987, 2014.

[5] VirusTotal, "Virus total," https://www.virustotal.com, 2021.

[6] D. Stefan, P. Buiras, E. Z. Yang, A. Levy, D. Terei, A. Russo, and D. Mazières, "Eliminating cache-based timing attacks with instruction-based scheduling," in *European Symposium on Research in Computer Security*. Springer, 2013, pp. 718–735.

[7] S. Guo, M. Wu, and C. Wang, "Adversarial symbolic execution for detecting concurrency-related cache timing leaks," in *ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*, 2018.

[8] M. Wu, S. Guo, P. Schaumont, and C. Wang, "Eliminating timing side-channel leaks using program repair," in *Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis*. ACM, 2018, p. 15–26.

[9] M. R. Gadelha, F. R. Monteiro, J. Morse, L. C. Cordeiro, B. Fischer, and D. A. Nicole, "ESBMC 5.0: An industrial-strength C model checker," in 33^{rd} *ACM/IEEE Int. Conf. on Automated Software Engineering (ASE'18)*. New York, NY, USA: ACM, 2018, pp. 888–891.

[10] M. Ganai, D. Lee, and A. Gupta, "DTAM: Dynamic taint analysis of multi-threaded programs for relevancy," in *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*. ACM, 2012, pp. 46–56.

[11] X. Zhang, Z. Yang, Q. Zheng, Y. Hao, P. Liu, L. Yu, and T. Liu, "Debugging multithreaded programs as if they were sequential," *IEEE Access*, vol. 6, no. 1, pp. 40 024–40 040, 2018.

[12] S. Guo, M. Kusano, and C. Wang, "Conc-iSE: Incremental symbolic execution of concurrent software," in *Automated Software Engineering (ASE), 2016 31st IEEE/ACM International Conference on*. IEEE, 2016, pp. 531–542.

[13] Microsoft, "Multitasking," https://bit.ly/3ayU5gt, 2021.

[14] "VX heaven," https://bit.ly/32AzHHw, 2021.

[15] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao, "Replacement attacks: automatically impeding behavior-based malware specifications," in *In International Conference on Applied Cryptography and Network Security*, 2015, pp. 497–517.

[16] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, "The SPLASH-2 programs: Characterization and methodological considerations," in *Computer Architecture, 1995. Proceedings., 22nd Annual International Symposium on*. IEEE, 1995, pp. 24–36.

[17] "Microsoft open source code," https://git.io/JOXnk, 2021.

[18] Microoft, "Red5," http://red5.org/, 2021.

[19] jseidl, "Goldeneye," https://github.com/jseidl/GoldenEye/, 2021.

[20] A. Software, "Jevereg," http://jevereg.amnpardaz.com/, 2021.

[21] "Falcon," https://www.reverse.it/, 2021.

[22] "Anlyz," https://sandbox.anlyz.io/, 2021.

[23] "Anyrun," https://app.any.run/, 2021.

[24] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, "Unveil: A large-scale, automated approach to detecting ransomware," in *USENIX Security Symposium*, 2016, pp. 757–772.

[25] D. Kirat, G. Vigna, and C. Kruegel, "BareCloud: Bare-metal analysis-based evasive malware detection." in *USENIX Security Symposium*, 2014, pp. 287–301.

[26] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, "Symbolic model checking without BDDs," *Tools and Algorithms for the Construction and Analysis of Systems*, pp. 193–207, 1999.

[27] N. Chong, B. Cook, K. Kallas, K. Khazem, F. R. Monteiro, D. Schwartz-Narbonne, S. Tasiran, M. Tautschnig, and M. R. Tuttle, "Code-level model checking in the software development workflow," in *2020 IEEE/ACM 42nd International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2020, pp. 11–20.

[28] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, "NetSMC: A custom symbolic model checker for stateful network verification," in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*. Santa Clara, CA: USENIX Association, Feb. 2020, pp. 181–200.

[29] D. Kroening and M. Tautschnig, "CBMC–C bounded model checker," in *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2014, pp. 389–391.

[30] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, "The software model checker blast," *International Journal on Software Tools for Technology Transfer*, vol. 9, no. 5-6, pp. 505–525, 2007.

[31] M. Said, C. Wang, Z. Yang, and K. Sakallah, "Generating data race witnesses by an SMT-based analysis," in *NASA Formal Methods Symposium*. Springer, 2011, pp. 313–327.

[32] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan, "Best: A symbolic testing tool for predicting multi-threaded program failures," in *Proceedings of the 2011 26th IEEE/ACM International Conference on Automated Software Engineering*. IEEE Computer Society, 2011, pp. 596–599.

[33] X. Zhang, Z. Yang, Q. Zheng, Y. Hao, P. Liu, and T. Liu, "Tell you a definite answer: Whether your data is tainted during thread scheduling," *IEEE Transactions on Software Engineering*, vol. 46, no. 9, pp. 916–931, 2020.

[34] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *International conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 2008, pp. 337–340.

[35] X. Zhang, Z. Yang, Q. Zheng, P. Liu, J. Chang, Y. Hao, and T. Liu, "Automated testing of definition-use data flow for multithreaded programs," in *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, March 2017, pp. 172–183.

[36] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, "Testing concurrent programs to achieve high synchronization coverage," in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*. ACM, 2012, pp. 210–220.

[37] C. Wang, M. Said, and A. Gupta, "Coverage guided systematic concurrency testing," in *Proceedings of the 33rd International Conference on Software Engineering*. ACM, 2011, pp. 221–230.

[38] J. Yu, S. Narayanasamy, C. Pereira, and G. Pokam, "Maple: a coverage-driven testing tool for multithreaded programs," *ACM SIGPLAN Notices*, vol. 47, no. 10, pp. 485–502, 2012.

[39] E. Rudd, A. Rozsa, M. Gunther, and T. Boult, "A survey of stealth malware: Attacks, mitigation measures, and steps toward autonomous open world solutions," *IEEE Communications Surveys & Tutorials*, 2016.

[40] M. Ramilli and M. Bishop, "Multi-stage delivery of malware," in *5th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 2010, pp. 91–97.

[41] S. Zhao, R. Gu, H. Qiu, T. O. Li, Y. Wang, H. Cui, and J. Yang, "OWL: Understanding and detecting concurrency attacks," in *48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2018, pp. 219–230.

[42] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *4th USENIX Workshop on Hot Topics in Parallelism (HotPar 12)*. Berkeley, CA: USENIX Association, Jun. 2012.

[43] M. Ramilli, M. Bishop, and S. Sun, "Multiprocess malware," in *Malicious and Unwanted Software (MALWARE), 2011 6th International Conference on*. IEEE, 2011, pp. 8–13.

[44] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: automatically evading system-call-behavior based malware detection," *Journal in Computer Virology*, vol. 8, no. 1, pp. 1–13, 2012.

[45] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee, "Jekyll on iOS: When benign apps become evil." in *Usenix Security*, vol. 13, 2013.

[46] M. W. Eichin and J. A. Rochlis, "With microscope and tweezers: An analysis of the internet virus of November 1988," in *Proceedings. 1989 IEEE Symposium on Security and Privacy*. IEEE, 1989, pp. 326–343.

[47] E. Kaspersky, "RMNS-the perfect couple," *Virus Bulletin*, pp. 8–9, 1995.

[48] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on Android," *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 6, pp. 1252–1264, June 2016.

[49] Y. Ji, Y. He, D. Zhu, Q. Li, and D. Guo, "A mulitiprocess mechanism of evading behavior-based bot detection approaches," in *International Conference on Information Security Practice and Experience*. Springer, 2014, pp. 75–89.