ConcSpectre: Be Aware of Forthcoming Malware Hidden in Concurrent Programs

Yang Liu, *Member, IEEE*, Zisen Xu, Ming Fan, *Member, IEEE*, Yu Hao, Kai Chen, *Member, IEEE*, Hao Chen, *Member, IEEE*, Yan Cai, *Member, IEEE*, Zijiang Yang, and Ting Liu, *Member, IEEE*

***Abstract*—Concurrent programs with multiple threads executing in parallel are widely used to unleash the power of multicore computing systems. Owing to their complexity, a lot of research focuses on testing and debugging concurrent programs. Besides correctness, we find that security can also be compromised by concurrency. In this article, we present concurrent program spectre (ConcSpectre), a new security threat that hides malware in nondeterministic thread interleavings. To demonstrate such threat, we have developed a stealth malware technique called concurrent logic bomb 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, we can activate such ConcSpectre malware with a very high probability (e.g., $>90\%$) by remotely disturbing thread scheduling. In the evaluation, more than 1000 ConcSpectre samples are generated, which bypassed most of the antivirus 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.**

***Index Terms*—Concurrent logic bomb (CLB), concurrent programs, concurrent program spectre (ConcSpectre), controllable probabilistic activation (CPA), software security.**

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 the 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 article, 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 nondeterminism 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 the 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 multicore 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 the paths is extremely hard in the era of multicore processors due to the nondeterminism of thread interleavings.

Manuscript received February 6, 2022; accepted March 22, 2022. This work was supported in part by the National Key R&D Program of China under Grant 2018YFB0803501, in part by the National Natural Science Foundation of China under Grant U1766215, 61772408, Grant 61632015, Grant 61833015, Grant 62002281, and Grant 61902306, in part by the China Postdoctoral Science Foundation under Grant 2019TQ0251, Grant 2020M683520, and Grant 2020M673439, in part by the Fundamental Research Funds for the Central Universities, the Youth Talent Support Plan of Xi'an Association for Science and Technology under Grant 095920201303, in part by the CCF–Tencent Open Research Fund, and in part by the 2020 Industrial Internet Innovation Development Project—Industrial Internet Penetration Testing and Crowdsourced Testing Platform under Grant TC200H01P. Associate Editor: Z. Zheng. *(Corresponding author: Ming Fan.)*

Yang Liu, Zisen Xu, Ming Fan, Zijiang Yang, and Ting Liu are with the Ministry of Education Key Laboratory for Intelligent Networks and Network Security, Xi'an Jiaotong University, Xi'an 710049, China (e-mail: yliu.xjtu@gmail.com; xzs05350332@stu.xjtu.edu.cn; mingfan@mail.xjtu.edu.cn; zijiang@xjtu.edu.cn; tingliu@mail.xjtu.edu.cn).

Yu Hao is with the Department of Computer Science and Engineering, University of California, Riverside, CA 92521 USA (e-mail: yhao016@ucr.edu).

Kai Chen is with the State Key Laboratory of Information Security, Institute of Information Engineering, Chinese Academy of Sciences, Beijing 100195, China, with the School of Cyber Security, University of Chinese Academy of Sciences, Beijing 100049, and also with the Beijing Academy of Artificial Intelligence, Beijing 100084, China (e-mail: chenkai@iie.ac.cn).

Hao Chen is with the Department of Computer Science, University of California, Davis, CA 95616 USA (e-mail: chen@ucdavis.edu).

Yan Cai is with the State Key Laboratory of Computer Science, Institute of Software, Chinese Academy of Sciences, Beijing 100190, China, and also with the School of Computer Science and Technology, University of Chinese Academy of Sciences, Beijing 100049, China (e-mail: ycai.mail@gmail.com).

Color versions of one or more figures in this article are available at https://doi.org/10.1109/TR.2022.3162694.

Digital Object Identifier 10.1109/TR.2022.3162694

```
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 multithreaded download program.

Consider the multithreaded program given in Fig. 1. The main thread creates two threads $T1$ and $T2$ to concurrently execute the function *multidownload* to download four files in total. The functions *is _ download* and *set _ download* are used to avoid downloading the same file more than once. A typical execution trace is $\pi 1$: T1(*download*(0))–T2(*download*(1))–T1(*download*(2))–T2(*download*(3)). However, $\pi 1$ 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 $\pi 2$: T1(*download*(0))–T2(*download*(1))–T2(*download*(2))–T1(*download*(3)). A malware analysis on $\pi 1$ and $\pi 2$ may report different results. The behavior of a concurrent program relies on not only its inputs but also the thread scheduling. The inherent nondeterminism of multithreaded 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 challenge is to snugly hide a malware sample in a concurrent program while bypassing the malicious behavior detection of modern malware detection tools. The second challenge is to trigger the malicious behavior effectively in a controllable manner. This article 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 1000 malware samples have been generated, which evade the detection of most of the antivirus 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 article makes the following contributions.

1) 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.
2) 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 most of the state-of-the-art dynamic and static detectors.
3) 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.

The rest of this article is organized as follows. Section II explains the basic idea of ConcSpectre malware through a motivating example. Sections III and IV give a detailed explanation of CLB and CPA techniques, respectively. Section V shows the performance of ConcSpectre malware against well-known online malware detection systems. Section VI discusses the threats of ConcSpectre and corresponding detection and defense strategies. Section VII introduces the related work. Finally, Section VIII concludes this article.

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

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

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

(a)

```
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)

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

(c)

Fig. 2. Demo case of ConcSpectre. (a) Snippet of malware. (b) ConcSpectre Malware. (c) Host program of ConcSpectre.

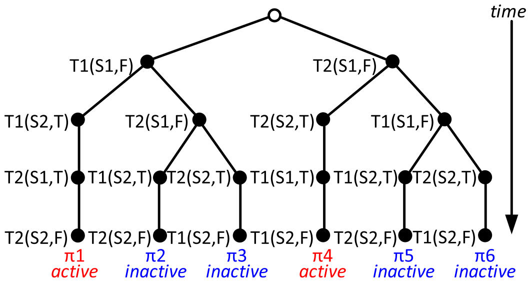


Fig. 3. Execution traces of ConcSpectre malware in Fig. 2. The last row indicates whether the malicious behavior is activated.

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 $\pi 1$ and $\pi 4$. It can be observed that the malicious behavior manifests in $\pi 1$ 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 1680 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. First, 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. Second, ConcSpectre can be applied to launch various large-scale attacks, such as botnet and worm. Specifically, although botnets use existing common protocols like Internet Relay Chat, bots within a botnet are highly similar due to the preprogrammed activities related to the same Command and Control server. Therefore, current botnet detection methods mainly detect botnets based on spatial–temporal correlation in network traffic [6], [7]. ConcSpectre bots in a botnet could hide their abnormal behaviors well by randomly activating once in thousands of runs, which breaks the correlation of bots with acceptable performance loss.

B. Basic Assumptions

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

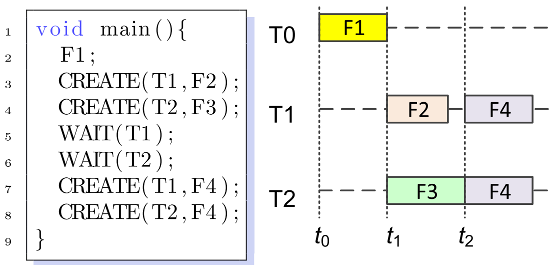
First, a piece of malware can be partitioned and each component is not detectable by current malware analysis techniques. Since each component by itself does not cause any harm, its behavior is usually not suspicious. In Section V-D, we partition four real malware samples into many components to escape from malware detection engines.

Second, ConcSpectre malware can be installed on a victim's system that supports multithreading using various methods (e.g., phishing, 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.

Third, 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 Section V-C, we demonstrate how to remotely activate 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 article, we partition a malware sample manually with the following consideration: 1) automated code partition has been a difficult problem for decades and 2) with domain and code knowledge, an attacker may give a partition trickier



```
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.

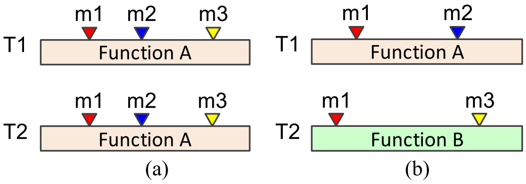


Fig. 5. Malware section injection methods. (a) SPE. (b) CPE.

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 host functions, we classify the functions in a concurrent program into three types, as illustrated in Fig. 4.

1) *Nonparallel-execution (NPE)* functions cannot be executed with any other functions simultaneously (e.g., F1).
2) *Cross-parallel-execution (CPE)* functions can be executed with other functions concurrently (e.g., F2 and F3).
3) *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 dependence among *malware sections*, such type of dependence cannot be exploited by malware analysis tools. We will discuss this in Section III-B.

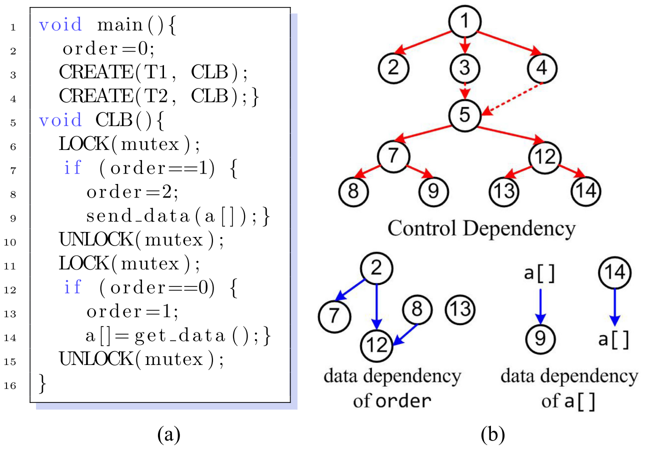
When we inject *malware components* into CPE functions, the injection positions of all sections are different. As shown in Fig. 5(b), ($m1$) is injected into two CPE functions, and ($m2$ and $m3$) is only in one CPE function. When these two CPE functions are invoked in parallel, three *malware sections* 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 the 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. There are two major challenges for CPE functions. First, a single injection location in an SPE function indicates the concurrency of multiple threads. In comparison, we need to consider multiple injection locations for CPE functions simultaneously, which requires more changes to the source code and makes the injection less uniform. Second, it is complicated to perturb the interleaving of CPE functions, which makes probabilistic activation more difficult. Thus, we mainly focus on SPE functions in our work. The feasibility of CPE functions for malware injection will be demonstrated in Section V-F.

B. Stealthiness of CLB

There are two common types of malware detection techniques: 1) static analysis techniques attempt to identify malicious code by searching suspicious strings or blocks of code and 2) dynamic analysis techniques seek to identify malicious behaviors after deploying and executing the samples. Since the CLB technique partitions the malware and hides its components into various places in a host program, it is almost impossible for static analysis techniques to detect malware, as confirmed by the experiments in Section V-D. In this section, we will show that the CLB technique could also evade current dynamic analysis techniques.

Dynamic malware detection exploits control dependence and data dependence 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 dependence of



```
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 }
```

Fig. 6. Control and data dependence analysis. (a) Code snippet. (b) Program dependence.

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 dependence 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 dependence and data dependence, there is no execution path containing the *get-send* pattern. Therefore, the dependencies could not guide dynamic analysis to defend CLB. Moreover, we envision that 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* [8]–[10].

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 [11]) or symbolic execution (e.g., DTAM [12], Proactive-Debugger [13], and Conc-iSE [14]). 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, nonlinear computation, and the sheer size of interleavings. Moreover, as source codes are usually unavailable, it would be more difficult to detect malicious behaviors when antianalysis techniques, such as ambiguous translation [15], are introduced.

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

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

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

As demonstrated by Fig. 3, the triggering condition in (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, (1), which is valid for sequential programs, is no longer valid for concurrent programs. We define *probabilistic activation* for concurrent malware as

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

Equation (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 CPA as follows:

$$P(ConcS = active|in \in IN_{ACT}) = \theta$$
$$P(ConcS = active|in \in IN_{ACT} \wedge side\_cond) = \gamma \qquad (3)$$

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 that the activation strategy is revised to *section 1 should be executed before section 2* ($S_1 < S_2$), the 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.

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
$\pi 1$	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
$\pi 2$	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
$\pi 3$	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
$\pi 4$	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
$\pi 5$	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
$\pi 6$	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

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 article, we have analyzed nine situations, including two to four threads and two to four *malware sections* per thread. The number of malware-activated thread interleavings is shown in Table II. With activation strategy "$S_1 < S_2 < S_3$," the malicious behavior would always be triggered. However, with "$S_3 < S_2 < S_1$," only 0, and 14% thread interleavings would trigger the malicious behavior when there are two to four 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 CPA on Windows

According to (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.

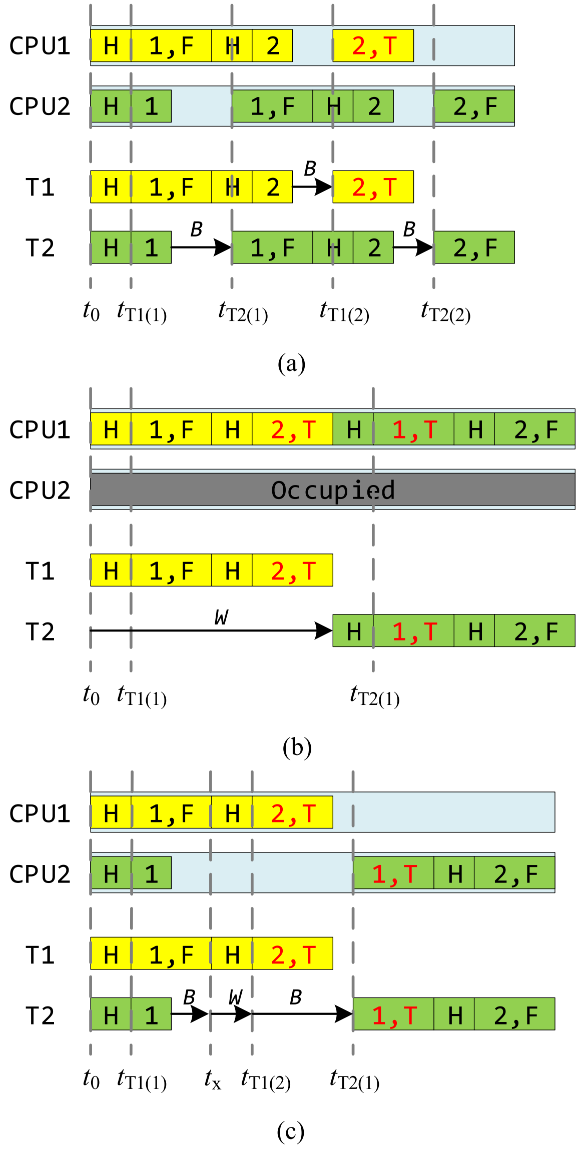
On Windows OS, the scheduler divides the available processor time in a round-robin fashion among the processes or threads following scheduling priority. Thus, there are three variable factors to influence thread scheduling: *available processor time*, *scheduling priority*, and *round-robin mechanism*. 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 $\pi 2$ 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 Fig. 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 8-GB memory. We create four threads to execute four functions (S_1–S_4) that read and write some local files with the same order. We simulate nine groups of experiments, where each group represents a different CPU usage. In each group, the concurrent program runs 10 000 times, and we execute zero to eight programs with infinite loops to simulate the system load from 0 to 100%. All the 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 the CPU usage rises. In particular, the occurrence rate of the fully-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%.



(a)

(b)

(c)

Fig. 7. Thread interleavings (1,T/1,F means *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). (a) Low system load. (b) High system load. (c) $t_{\text{Host}} < t_{\text{Get\_CPU}}$.

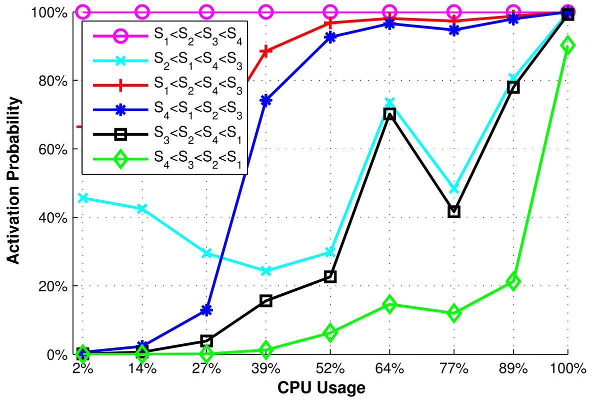


Fig. 8. Activation probability under different activation strategies.

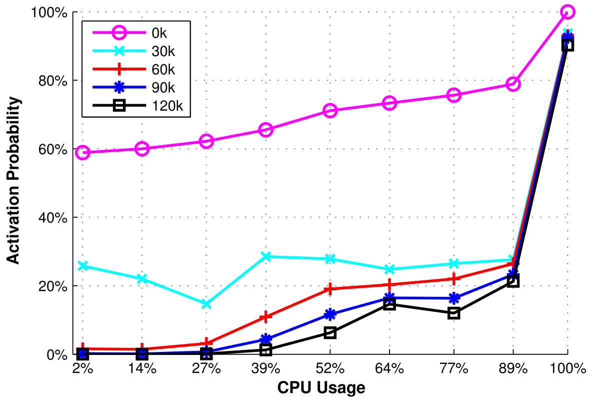


Fig. 9. Activation probability of $S_4 < S_3 < S_2 < S_1$.

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 ms) for each thread it executes. The running thread is suspended when its *time slice* elapses, allowing another thread to run [16]. 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, which could be executed very quickly. Like the settings in Fig. 8, we run the program 10 000 times for nine different CPU usages, respectively. Experimental results show that the occurrence rates of fully-reverse-order sequence $S_4 < S_3 < S_2 < S_1$ (i.e., the pink line marked as 0k) in Fig. 9 are all over 60%, which are quite different from those (i.e., the blue line marked as $S_4 < S_3 < S_2 < S_1$) in Fig. 8. To further analyze this phenomenon, 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*, where the numbers of loops are 30k, 60k, 90k, and 120k, respectively. 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.

TABLE III
MALWARE SAMPLES

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

LOC: Lines of code.

TABLE IV
CONCURRENT PROGRAMS

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

LOC: Lines of code. #Function, #SPE, and #Injection point are the number of functions, SPE functions, and injection points, respectively.

V. EVALUATION

A. Experimental Setup

In this section, we select four widely studied open-source malware samples from VX Heavens [17], including *BullMoose*, *Branko*, *Hunatcha*, and *Hunatchab*, as shown in Table III[1] [18]. Specifically, BullMoose is a basic HTML File infector that changes a registry key to run this program instead of iexplore.exe. It appends the infection string (i.e., a Javascript code snippet) into the HTML file and opens it with iexplore.exe by calling the ShellExecute() API. It is a proof-of-concept Trojan to demonstrate the security risk of Microsoft Windows systems with Internet Explorer. *Branko* attempts to spread using a predefined users/password list and runs Taskkill to shut down antivirus programs. *Hunatcha* is a worm that spreads via peer-to-peer software by infecting files and modifying registry keys. *Hunatchab* is a variant of *Hunatcha*.

As shown in Table IV, we select nine programs from the well-known concurrent testing benchmark SPLASH [19] and one program from Microsoft Open Source Code [20] as the host programs. By analyzing their source codes, we identify 217 SPE functions from these programs. Then, we select 1–15 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. All four malware samples are hidden into benign concurrent programs to construct ConcSpectre and verify its stealthiness against current malware analysis techniques. Besides, *BullMoose* is selected as the demo sample to demonstrate how to generate and remotely trigger ConcSpectre malware.

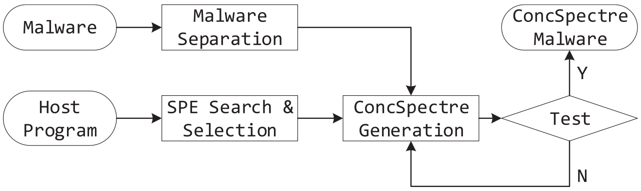


Fig. 10. Workflow of ConcSpectre generation.

In this article, we generate over 1000 samples of ConcSpectre 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 VirusTotal and four dynamic malware analysis systems to demonstrate: 1) how to generate real ConcSpectre malware samples; 2) how to trigger ConcSpectre malware remotely; 3) whether antivirus systems can detect ConcSpectre malware; and 4) whether dynamic malware analysis systems can detect ConcSpectre malware.

Note that the resource consumption of concurrent programs is usually high. For example, programs in the SPLASH benchmark will be executed within 0.1 s to a few minutes according to the size of the input set, and the memory footprint is at the level of megabytes or gigabytes [21]. In contrast, malware is usually lightweight for better stealthiness. Specifically, most *malicious sections* could be executed within a few milliseconds at a very low memory footprint (i.e., at the level of kilobytes). Therefore, the side effects of malware injection on concurrent host programs are usually insignificant.

B. ConcSpectre Malware Generation

Fig. 10 shows 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.

1) malware=*BullMoose*, host program=*fft*.
2) 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 system load (CPU usage is higher than 75%).
3) A global variable is used to control the execution of each *malware section*.
4) 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 dependence and data dependence of malware. These dependencies are used to guide malware partition to make sure that the relation between different malware fragments is as little as possible. Then, each component is checked with various antimalware systems. If there are any

[1] As bots are essentially Trojan/worm combinations belonging to a large malicious network (i.e., a botnet), the detection results of an individual bot based on static and dynamic malware analysis could be represented by Trojan/worm malware samples. Further discussion on traffic-based botnet detection is beyond the scope of this article.

TABLE V
MAIN CONCSPECTRE MALWARE SAMPLES FOR TEST

Injected Malware	Activation Orders	#Samples	#Total Samples
BullMoose	$S1 < S2 < S3 < S4$	9	298
	$S4 < S1 < S2 < S3$	1	
	$S4 < S3 < S2 < S1$	77	
	$S4 < S1 < S3 < S2$	77	
	$S4 < S2 < S3 < S1$	77	
	$S1 < S3 < S4 < S2$	19	
	$S1 < S4 < S3 < S2$	19	
	$S1 < S2 < S4 < S3$	19	
Branko	$S4 < S3 < S2 < S1$	77	231
	$S4 < S1 < S3 < S2$	77	
	$S4 < S2 < S3 < S1$	77	
Hunatcha	$S4 < S3 < S2 < S1$	77	231
	$S4 < S1 < S3 < S2$	77	
	$S4 < S2 < S3 < S1$	77	
Hunatchab	$S4 < S3 < S2 < S1$	77	231
	$S4 < S1 < S3 < S2$	77	
	$S4 < S2 < S3 < S1$	77	

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 that each component would not be classified as a malicious program. A global variable is added to control the execution order of the malware components.[2]

In the *SPE Search & Selection* module, we search the source code of the host program to find all possible SPE 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 first select a control strategy guided by the rate of malware-activated trace (e.g., Table 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. As shown in Table V, by injecting four malware into 77 injection points of all host programs, we have generated 991 representative ConcSpectre malware samples with different activation orders for further test.

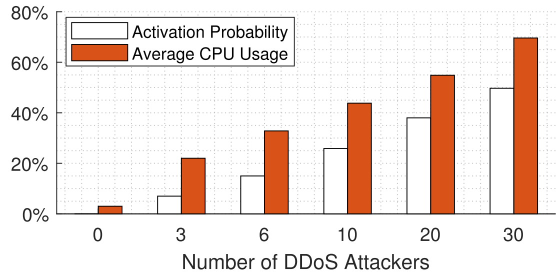


Fig. 11. CPU usage and activation probability.

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.

C. Remote Triggering of 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 [22] 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.

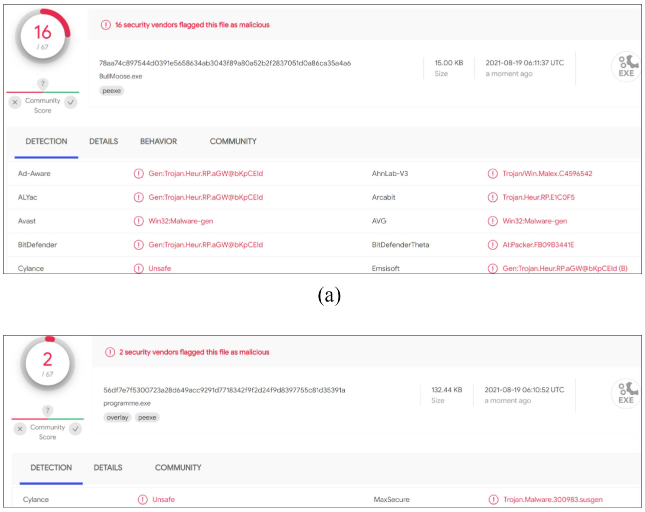
To simulate the real case with large-scale concurrent accesses, we run a real-world HTTP DDoS test tool GoldenEye [23] to visit our web server and configure it to work under five different groups of users: 3, 6, 10, 20, and 30. Each user visits the web server with ten concurrent sockets. Note that these 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 7% and 51%, respectively.

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

D. ConcSpectre Versus Antivirus System

One design requirement of ConcSpectre malware is to escape from the antivirus systems. Therefore, we select all antivirus engines from VirusTotal to analyze generated ConcSpectre malware. Note that these engines are well configured and also up

[2] The code of four *malware sections* is provided on https://git.io/ConcSpectre. Moreover, a VirtualBox environment is available on https://bit.ly/ConcSpectreVM.



16 / 67

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

(a)

2 / 67

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)

Fig. 12. Detection results of VirusTotal. (a) BullMoose. (b) BullMoose-fft-C1.

to date. The set includes widely used ones such as McAfee, Microsoft, and Symantec.

We submit the executable file of *BullMoose* and *BullMoose-fft-C1* 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) and 2) only two engines (”Cylance” and “MaxSecure”) report anomalies from the ConcSpectre samples in Fig. 12(b). After further analysis, these two alarms are both false positives, which are inherited from the host program *fft*.

To justify whether the ConcSpectre can bypass various antivirus engines, we choose 924 samples with the following three activation strategies: C2 ($S4 < S3 < S2 < S1$), C3 ($S4 < S1 < S3 < S2$), and C4 ($S4 < S2 < S3 < S1$) and submit them to VirusTotal. The results are similar to the previous one. The original malware samples could be detected by most of the engines. Meanwhile, only a few engines could detect ConcSpectre samples correctly. This exposes the potential threat of ConcSpectre.

E. ConcSpectre Versus 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* [24], *Falcon* [25], *Anlyz* [26], and *ANYRUN* [27]. They are developed on different well-known sandboxes [12], [28], [29].

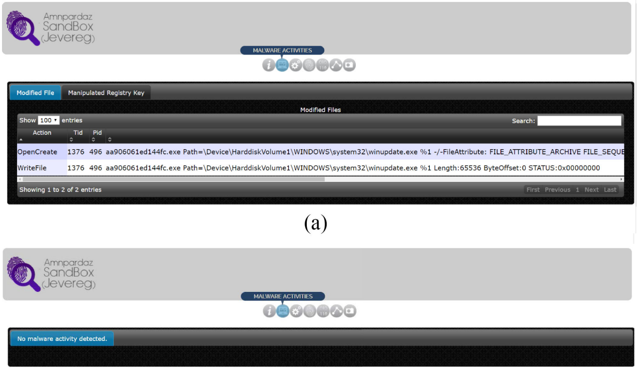
Initially, we select the ConcSpectre samples generated from the *BullMoose* in Table VI. Since the numbers of SPE functions in different host programs are different, the number of malware samples in each host program is also different (listed as #Sample). One special variant is generated with the activation 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*.

Together with *Bullmoose*, we submit 233 programs to four dynamic malware analysis systems. As shown in Table VI, *Bullmoose* and *BullMoose-fft-A* have been detected by all systems.

TABLE VI
DETECTION RESULTS OF DYNAMIC MALWARE ANALYSIS

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
BullMoose-*-C in total (231)	150	0	0	87
Detection Rate of BullMoose-*-C	(64.94%)	(0%)	(0%)	(37.66%)

Three different activation strategies C2, C3, and C4 are marked as -C, the strategy $S1 < S2 < S3 < S4$ is marked as -A, and #Sample is the number of malware samples.



Modified File | Manipulated Registry Key

Modified Files

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

(a)

No malware activity detected.

(b)

Fig. 13. Detection results of Jevereg. (a) Jevereg report BullMoose-ocean_nc-C2. (b) Jevereg report BullMoose-ocean_nc-C5.

For the ConcSpectre samples, the results are different. In particular, 150 samples from six 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 Section IV-C. If our conjecture is true, *Jevereg* and *ANYRUN* would fail to 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 another ten ConcSpectre malware samples containing this variant. From Table VII, no

TABLE VII
DETECTION RESULTS OF TEN CONCSPECTRE MALWARE SAMPLES WITH HYBRID ACTIVATION STRATEGY $C5$ (THE ORIGINAL MALWARE: BULLMOOSE)

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 VIII
DETECTION RESULTS OF 33 MALWARE SAMPLES, WHERE THREE ARE THE ORIGINAL MALWARE SAMPLES, AND THE OTHER 30 ARE THE GENERATED CONCSPECTRE MALWARE SAMPLES WITH HYBRID ACTIVATION STRATEGY $C5$

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

#Sample is the number of malware samples; the original malware: Branko, Hunatcha, Hunatchab.

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.

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 generate one ConcSpectre sample for each host program and malware, in total 30. Together with three original malware samples, 33 samples are submitted to four analysis systems, respectively. The detection results are shown in Table VIII.

From Tables VII and VIII, all four original malware samples would be detected as malware by four dynamic analysis systems. In contrast, none of the 40 generated ConcSpectre malware samples with the hybrid activation sequence $C5$ have been detected. Therefore, by introducing elaborate parallel-execution requirement into the activation sequence, ConcSpectre can escape from dynamic malware analysis techniques under most system load conditions.

F. ConcSpectre Malware With CPE Functions

In the previous experiments, we have constructed in total 1031 ConcSpectre malware samples with the SPE function, i.e., 991 samples in Table V, ten samples in Table VII, and 30 samples in Table VIII. In most concurrent programs, there are more CPE functions than SPE functions. Attackers can use a similar method to partition malware and inject their components into CPE functions to bypass the static malware detection. Here, we will construct and test CPE-function-targeted ConcSpectre malware.

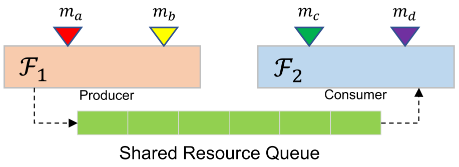


Fig. 14. Demo of CPE-function-targeted ConcSpectre malware sample, whose order of *malware sections* is represented by $(m_a m_b,\ m_c m_d)$.

TABLE IX
ACTIVATION PROBABILITY OF CPE-FUNCTION-TARGETED CONCSPECTRE MALWARE WITH DIFFERENT INJECTION ORDERS UNDER DIFFERENT SYSTEM LOADS

Case	Injection Order	Activation Probability (%)	
		Low CPU Usage	High CPU Usage
1	$(m_1 m_2 m_3,\ m_4)$	0.00	51.50
2	$(m_1 m_2,\ m_3 m_4)$	0.10	74.00
3	$(m_1 m_3,\ m_2 m_4)$	70.40	80.40
4	$(m_4 m_3,\ m_2 m_1)$	0.00	0.00

As shown in Fig. 14, we divide the *BullMoose* malware into four sections (i.e., m_1, m_2, m_3, and m_4) and inject them into two CPE functions with different orders. Considering the race condition, we utilize a simplified producer–consumer model with a *producer* function $\mathcal{F}_1$ and a *consumer* function $\mathcal{F}_2$. The *producer* ($\mathcal{F}_1$) will add stuff into a shared buffer concurrently, and the *consumer* ($\mathcal{F}_2$) will take stuff from the shared buffer. Each injection order is represented by a tuple. For example, $(m_a m_b,\ m_c m_d)$ indicates that m_a is injected before m_b in $\mathcal{F}_1$, and m_c is injected before m_d in $\mathcal{F}_2$.

First, we will analyze how the threads of CPE functions are scheduled under different system load conditions. When the system load is low, there are multiple free CPU units. In this condition, the thread scheduler would prefer to assign different CPU units to different functions' threads. To analyze the impact of thread scheduling, we add a 9120k-empty-loop in each *malware section* so that the thread could execute across multiple pieces of time slicing. Therefore, the thread switching would occur frequently during the execution process. When the system load is high, the functions' threads would be executed sequentially on the same CPU unit. In this condition, this CPU unit would prefer to finish executing $\mathcal{F}_1$ first before executing $\mathcal{F}_2$.

For simplification, we select some typical injection orders for two CPE functions and create two threads for each CPE function. Then, we run the program 1000 times for each injection order and record the activation probability of malware under different system loads (low CPU usage: $< 20\%$; high CPU usage: $\approx 100\%$). The experimental results in Table IX can be illustrated case by case as follows.

1) *Case 1:* The expected execution trace is $\mathcal{F}_1^{m_1,m_2,m_3} - \mathcal{F}_2^{m_4}$. When the system load is low, it would be difficult to execute $\mathcal{F}_1^{m_1,m_2,m_3}$ due to the frequent thread switching. Thus, the malware could hardly be activated (i.e.,

0%). When the system load is high, it is much easier to execute $\mathcal{F}_1^{m_1,m_2,m_3}$ as all functions tend to be executed sequentially. As the possibilities of executing $\mathcal{F}_1 - \mathcal{F}_2$ and $\mathcal{F}_2 - \mathcal{F}_1$ are almost the same, the final activation probability would be close to 50% (i.e., 51.5%).

2) *Case 2:* The expected execution trace is $\mathcal{F}_1^{m_1,m_2} - \mathcal{F}_2^{m_3,m_4}$. This case is similar to case 1. Besides, the execution time for $\mathcal{F}_1^{m_1,m_2}$ or $\mathcal{F}_2^{m_3,m_4}$ would be lower than that of $\mathcal{F}_1^{m_1,m_2,m_3}$. Therefore, the activation probability of case 2 would be a little higher than case 1.

3) *Case 3:* The expected execution trace is $\mathcal{F}_1^{m_1} - \mathcal{F}_2^{m_2} - \mathcal{F}_1^{m_3} - \mathcal{F}_2^{m_4}$. When the system load is low, it is easy to trigger the trace of $\mathcal{F}_1 - \mathcal{F}_2$ as the thread switching occurs frequently. Thus, the activation probability is high (i.e., 74%). When the system load is high, it is more likely to execute $\mathcal{F}_1$ and $\mathcal{F}_2$ one by one. Thus, the activation probability would increase accordingly (i.e., 80.4%).

4) *Case 4:* The expected execution trace is $\mathcal{F}_2^{m_1} - \mathcal{F}_2^{m_2} - \mathcal{F}_1^{m_3} - \mathcal{F}_1^{m_4}$. Owing to the resource race between the *producer* and the *consumer* (e.g., resources should be produced before consumed), $\mathcal{F}_1$ should be executed first before $\mathcal{F}_2$. Therefore, it is almost impossible to trigger the trace of $\mathcal{F}_2 - \mathcal{F}_1$. Therefore, the activation probability would always be close to zero.

We have uploaded these cases to detection systems in Sections V-D and V-E. From the results, only five out of 67 engines in VirusTotal would report anomalies, and none of the dynamic malware analysis systems could detect the actual malicious behavior. Namely, the CPE-function-targeted ConcSpectre malware samples could escape from most detection engines in VirusTotal and four online dynamic malware detection systems.

From the above cases, CLB and CPA techniques could be transplanted to CPE functions. However, the thread interleavings of CPE functions are determined by more factors than SPE functions, since CPE functions would present different privileges, external conditions, synchronization constraints, etc. Here, case 2 is preferred for constructing CPE-function-targeted ConcSpectre malware. Meanwhile, it is not suitable to use the injection orders of cases 3 and 4, as their activation probabilities are similar under different system load conditions. In short, we need meticulous consideration when applying CPA techniques to CPE functions.

VI. Discussion

A. Threats of ConcSpectre

VirusTotal is the largest online antimalware scanning platform and has been widely used for labeling malware data [30]. Moreover, the four online dynamic malware analysis systems are well known and advanced. Therefore, we could conclude from Section V that ConcSpectre could bypass most state-of-the-art detection methods. As the antimalware methods are always evolving together with the malware, we cannot guarantee that the proposed ConcSpectre malware could escape from detection all the time. Thus, it is necessary to introduce new antianalysis techniques to enhance ConcSpectre continuously. Fortunately, the CLB and CPA techniques could be easily combined with other antianalysis techniques like malware obfuscation techniques [31].

As discussed in Section V-F, CPE functions could also be leveraged to construct ConcSpectre malware. This indicates greater threats as CPE functions are more common in concurrent programs. However, activation strategies for CPE functions are more complicated than SPE functions. To better understand the threats of ConcSpectre, we need further investigation on CPE-function-targeted ConcSpectre techniques.

B. Detection and Defense Strategies

It is intractable for analysts to deal with ConcSpectre malware, which might be widely used in multicore systems and multithreaded programs. In this section, we present and discuss several possible solutions for detection and defense.

1) Exhaustive Examination: It is well known that model checking is a powerful technique for exploring the whole state space for a program [32]–[34]. 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 [11], [35], [36]. We apply the latest version (2021) of ESBMC [11] to analyze *cholesky* under a fixed input. The verification of one input could not finish within 3600 s. 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 [37]–[39]. Its advantage lies in the capability of automatically finding intricate interleavings. It encodes an execution trace of a multithreaded 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 [40]. We reproduce the encoding method in [41] and use it to verify an assertion in *cholesky*. For an execution trace with 250k events, it spends 35 s 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.

2) Affecting Thread Scheduling: Various methods of affecting scheduler are applied in concurrent program testing to cover as many different interleavings as possible [42]–[44]. 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 tradeoff 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 Section V-E, the attackers can design various CPA strategies, which would not be easily activated under both low and high system loads. Thus, the defender can randomly assign the CPU resource during testing. It would increase the chance to detect the hidden malware.

3) Serialization: The simultaneous execution of many threads is the fundamental cause of ConcSpectre malware being activated while bypassing the detection. A compromised 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 multitask 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

A. *Stealth Malware Techniques*

Current stealth malware techniques could be classified into four types: rootkit, code mutation, antiemulation and targeting mechanism [45]. 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 inject 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 nondeterministic 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 multistage malware [46], concurrency attack [47], [48], and cooperative attack [49]–[51]. Some well-known computer viruses, such as Internet worm [52] and RMNS [53], are multistage, which achieve an attack by cooperating distributed tasks across multiple processes. Xu *et al.* [54] find that the intercomponent 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 [54]. 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.* [51] 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. However, the cooperative attack suffers from two limitations: 1) it is difficult for such malware to be applied in a real-time attack [55] and 2) any failure in any process will lead to the failure of the entire process [49]. 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.* [47], [48] discover that errors in concurrent programs can lead to concurrency attacks and first prove that concurrency attacks are indeed viable. 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.

B. *Malware Analysis Techniques*

Current malware analysis techniques can be roughly categorized into three groups: static analysis, dynamic analysis, and hybrid analysis (i.e., the combination of static analysis and dynamic analysis) [56].

The static malware analysis technique detects the patterns of malicious actions from semantic or structural properties of the program, including string signature [57], [58], byte-sequence n-grams [59], syntactic library call [58], [60], and system dependencies [61]–[64]. Static analysis is fast and easy to implement without the need of executing program samples. However, the effectiveness of static analysis is hampered by various obfuscation techniques [65]. Moser *et al.* [66] introduce a scheme on how to leverage code obfuscation to evade the detection of static analysis in the work. In most cases, analysts must disassemble the executables before conducting the static analysis. Ming *et al.* [67] proposed a Logic Oriented Opaque Predicate detection tool for obfuscated binary code, which can cooperate with existing malware defense strategies.

Instead of analyzing disassembled programs, the dynamic malware analysis technique detects the malicious behavior during or after the program execution. Techniques for dynamic analysis mainly include function call monitoring [68]–[71], model checking [72], [73], and information flow tracking [74]–[76]. In addition, Peng *et al.* [77] propose X-Force to investigate security applications by forcibly executing each branch of executable malicious binary. To counter dynamic analysis, attackers try to insert independent calls in the actual execution flow of malware binaries to evade detection [78], [79]. Xue *et al.* [80] propose a novel auditing antimalware method and develop a service-oriented malware generation system, which generates the malware-based software product line techniques and enables the attack via dynamic loading technique. As the execution orders of threads are nondeterministic, a failure in concurrent programs is usually hard to reproduce even under the same execution context. To accelerate the process of data race failure, Qiu *et al.* [81] propose a stress testing method by controlling three influencing factors, namely, memory limitation, concurrency level, and parallel level. This technique could be leveraged to accelerate the process of malware detection in concurrent programs.

VIII. Conclusion

In this article, we presented 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 1000 ConcSpectre malware samples were generated based on four real malicious programs and ten concurrent programs. They can bypass most of the antivirus engines in VirusTotal and four online dynamic malware detection systems. Experimental results revealed the threat of ConcSpectre, which calls for an urgent redesign of malware detection techniques for concurrent programs.

After exploring several software testing techniques for detecting ConcSpectre malware, we found that 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 the future, we will further study detection and defense techniques against ConcSpectre with less performance loss.

References

[1] M. Lipp *et al.*, “Meltdown: Reading kernel memory from user space,” in *27th USENIX Secur. Symp. (USENIX Security 18)*, 2018, pp. 973–990.

[2] P. Kocher *et al.*, “Spectre attacks: Exploiting speculative execution,” in *Proc. IEEE Symp. Secur. Privacy*, 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 Commun. Surv. Tuts.*, vol. 16, no. 2, pp. 961–987, Second Quarter 2014.

[5] VirusTotal, “Virus total,” 2021. [Online]. Available: https://www.virustotal.com

[6] G. Gu, R. Perdisci, J. Zhang, and W. Lee, “BotMiner: Clustering analysis of network traffic for protocol- and structure-independent botnet detection,” in *Proc. 17th Conf. Secur. Symp.*, 2008, pp. 139–154.

[7] Y. Zhao *et al.*, “BotGraph: Large scale spamming botnet detection,” in *Proc. 6th USENIX Symp. Netw. Syst. Des. Implementation*, 2009, vol. 9, pp. 321–334.

[8] D. Stefan *et al.*, “Eliminating cache-based timing attacks with instruction-based scheduling,” in *Proc. Eur. Symp. Res. Comput. Secur.*, 2013, pp. 718–735.

[9] S. Guo, M. Wu, and C. Wang, “Adversarial symbolic execution for detecting concurrency-related cache timing leaks,” in *Proc. 26th ACM Joint Eur. Softw. Eng. Conf. Symp. Found. Softw. Eng.*, 2018, pp. 377–388.

[10] M. Wu, S. Guo, P. Schaumont, and C. Wang, “Eliminating timing side-channel leaks using program repair,” in *Proc. 27th ACM SIGSOFT Int. Symp. Softw. Testing Anal.*, 2018, pp. 15–26.

[11] 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 *Proc. 33rd ACM/IEEE Int. Conf. Autom. Softw. Eng.*, 2018, pp. 888–891.

[12] M. Ganai, D. Lee, and A. Gupta, “DTAM: Dynamic taint analysis of multi-threaded programs for relevancy,” in *Proc. ACM SIGSOFT 20th Int. Symp. Found. Softw. Eng.*, 2012, pp. 46–56.

[13] X. Zhang *et al.*, “Debugging multithreaded programs as if they were sequential,” *IEEE Access*, vol. 6, pp. 40024–40040, 2018.

[14] S. Guo, M. Kusano, and C. Wang, “Conc-iSE: Incremental symbolic execution of concurrent software,” in *Proc. 31st IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2016, pp. 531–542.

[15] C. Jung, D. Kim, W. Wang, Y. Zheng, K. H. Lee, and Y. Kwon, “Defeating program analysis techniques via ambiguous translation,” in *Proc. 36th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2021, pp. 1–6.

[16] Multitasking, Microsoft, Albuquerque, NM, USA, 2021. [Online]. Available: https://bit.ly/3ayU5gt

[17] “VX heaven,” 2021. [Online]. Available: https://bit.ly/32AzHHw

[18] J. Ming, Z. Xin, P. Lan, D. Wu, P. Liu, and B. Mao, “Replacement attacks: Automatically impeding behavior-based malware specifications,” in *Proc. Int. Conf. Appl. Cryptography Netw. Secur.*, 2015, pp. 497–517.

[19] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta, “The SPLASH-2 programs: Characterization and methodological considerations,” in *Proc. 22nd Annu. Int. Symp. Comput. Archit.*, 1995, pp. 24–36.

[20] “Microsoft open source code,” 2021. [Online]. Available: https://git.io/JOXnk

[21] P. Group *et al.*, “A memo on exploration of SPLASH-2 input sets,” *Princeton Univ.*, Princeton, NJ, USA, 2011. [Online]. Available: https://parsec.cs.princeton.edu/doc/memo-splash2x-input.pdf

[22] Microsoft, Albuquerque, NM, USA, 2021. [Online]. Available: http://red5.org/

[23] jseidl, “Goldeneye,” 2021. [Online]. Available: https://github.com/jseidl/GoldenEye/

[24] A. Software, “Jevereg,” 2021. [Online]. Available: http://jevereg.amnpardaz.com/

[25] “Falcon,” 2021. [Online]. Available: https://www.reverse.it/

[26] “Anlyz,” 2021. [Online]. Available: https://sandbox.anlyz.io/

[27] “Anyrun,” 2021. [Online]. Available: https://app.any.run/

[28] A. Kharraz, S. Arshad, C. Mulliner, W. K. Robertson, and E. Kirda, “Unveil: A large-scale, automated approach to detecting ransomware,” in *Proc. USENIX Secur. Symp.*, 2016, pp. 757–772.

[29] D. Kirat, G. Vigna, and C. Kruegel, “BareCloud: Bare-metal analysis-based evasive malware detection,” in *Proc. USENIX Secur. Symp.*, 2014, pp. 287–301.

[30] S. Zhu, Z. Zhang, L. Yang, L. Song, and G. Wang, “Benchmarking label dynamics of VirusTotal engines,” in *Proc. ACM SIGSAC Conf. Comput. Commun. Secur.*, 2020, pp. 2081–2083.

[31] P. O’Kane, S. Sezer, and K. McLaughlin, “Obfuscation: The hidden malware,” *IEEE Secur. Privacy*, vol. 9, no. 5, pp. 41–47, Sep./Oct. 2011.

[32] A. Biere, A. Cimatti, E. Clarke, and Y. Zhu, “Symbolic model checking without BDDs,” in *Tools Algorithms Construction Anal. Syst.*, 1999, pp. 193–207.

[33] N. Chong *et al.*, “Code-level model checking in the software development workflow,” in *Proc. IEEE/ACM 42nd Int. Conf. Softw. Eng.: Softw. Eng. Pract.*, 2020, pp. 11–20.

[34] Y. Yuan, S.-J. Moon, S. Uppal, L. Jia, and V. Sekar, “NetSMC: A custom symbolic model checker for stateful network verification,” in *Proc. 17th USENIX Symp. Netw. Syst. Des. Implementation*, Feb. 2020, pp. 181–200.

[35] D. Kroening and M. Tautschnig, “CBMC-C bounded model checker,” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2014, pp. 389–391.

[36] D. Beyer, T. A. Henzinger, R. Jhala, and R. Majumdar, “The software model checker blast,” *Int. J. Softw. Tools Technol. Transfer*, vol. 9, nos. 5/6, pp. 505–525, 2007.

[37] M. Said, C. Wang, Z. Yang, and K. Sakallah, “Generating data race witnesses by an SMT-based analysis,” in *Proc. 3rd Int. Conf. NASA Formal Methods*, 2011, pp. 313–327.

[38] M. K. Ganai, N. Arora, C. Wang, A. Gupta, and G. Balakrishnan, “BEST: A symbolic testing tool for predicting multi-threaded program failures,” in *Proc. 26th IEEE/ACM Int. Conf. Autom. Softw. Eng.*, 2011, pp. 596–599.

[39] 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 Trans. Softw. Eng.*, vol. 46, no. 9, pp. 916–931, Sep. 2020.

[40] L. De Moura and N. Bjørner, “Z3: An efficient SMT solver,” in *Proc. Int. Conf. Tools Algorithms Construction Anal. Syst.*, 2008, pp. 337–340.

[41] X. Zhang *et al.*, “Automated testing of definition-use data flow for multi-threaded programs,” in *Proc. IEEE Int. Conf. Softw. Testing, Verification Validation*, Mar. 2017, pp. 172–183.

[42] S. Hong, J. Ahn, S. Park, M. Kim, and M. J. Harrold, “Testing concurrent programs to achieve high synchronization coverage,” in *Proc. Int. Symp. Softw. Testing Anal.*, 2012, pp. 210–220.

[43] C. Wang, M. Said, and A. Gupta, “Coverage guided systematic concurrency testing,” in *Proc. 33rd Int. Conf. Softw. Eng.*, 2011, pp. 221–230.

[44] 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.

[45] E. Rudd, A. Rozsa, M. Gunther, and T. Boult, “A survey of stealth malware: Attacks, m autonomous open world solutions,” *IEEE Commun. Surv. Tuts.*, vol. 19, no. 2, pp. 1145–1172, Second Quarter 2017.

[46] M. Ramilli and M. Bishop, “Multi-stage delivery of malware,” in *Proc. 5th Int. Conf. Malicious Unwanted Softw.*, 2010, pp. 91–97.

[47] S. Zhao *et al.*, "OWL: Understanding and detecting concurrency attacks," in *Proc. 48th Annu. IEEE/IFIP Int. Conf. Dependable Syst. Netw.*, 2018, pp. 219–230.

[48] J. Yang, A. Cui, S. Stolfo, and S. Sethumadhavan, "Concurrency attacks," in *Proc. 4th USENIX Workshop Hot Topics Parallelism*, Berkeley, CA: USENIX Association, Jun. 2012, pp. 1–7. [Online]. Available: https://www.usenix.org/conference/hotpar12/workshop-program/presentation/yang

[49] M. Ramilli, M. Bishop, and S. Sun, "Multiprocess malware," in *Proc. 6th Int. Conf. Malicious Unwanted Softw.*, 2011, pp. 8–13.

[50] W. Ma, P. Duan, S. Liu, G. Gu, and J.-C. Liu, "Shadow attacks: Automatically evading system-call-behavior based malware detection," *J. Comput. Virol.*, vol. 8, no. 1, pp. 1–13, 2012.

[51] T. Wang, K. Lu, L. Lu, S. P. Chung, and W. Lee, "Jekyll on iOS: When benign apps become evil," in *Proc. 22nd Usenix Conf. Secur.*, vol. 13, 2013, pp. 559–572.

[52] M. W. Eichin and J. A. Rochlis, "With microscope and tweezers: An analysis of the internet virus of Nov. 1988," in *Proc. IEEE Symp. Secur. Privacy*,1989, pp. 326–343.

[53] E. Kaspersky, "RMNS-the perfect couple," *Virus Bull.*, vol. 7, pp. 8–9, May 1995. [Online]. Available: https://papers.vx-underground.org/papers/vb/199505.pdf

[54] K. Xu, Y. Li, and R. H. Deng, "ICCDetector: ICC-based malware detection on android," *IEEE Trans. Inf. Forensics Secur.*, vol. 11, no. 6, pp. 1252–1264, Jun. 2016.

[55] Y. Ji, Y. He, D. Zhu, Q. Li, and D. Guo, "A mulitiprocess mechanism of evading behavior-based bot detection approaches," in *Proc. Int. Conf. Inf. Secur. Pract. Experience*, 2014, pp. 75–89.

[56] S. S. Chakkaravarthy, D. Sangeetha, and V. Vaidehi, "A survey on malware analysis and mitigation techniques," *Comput. Sci. Rev.*, vol. 32, pp. 1–23, 2019.

[57] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, and R. E. Bryant, "Semantics-aware malware detection," in *Proc. IEEE Symp. Secur. Privacy*, 2005, pp. 32–46.

[58] A. H. Sung, J. Xu, P. Chavez, and S. Mukkamala, "Static analyzer of vicious executables (SAVE)," in *Proc. 20th Annu. Comput. Secur. Appl. Conf.*, 2004, pp. 326–334.

[59] W.-J. Li, K. Wang, S. J. Stolfo, and B. Herzog, "Fileprints: Identifying file types by n-gram analysis," in *Proc. 6th Annu. IEEE SMC Inf. Assurance Workshop*, 2005, pp. 64–71.

[60] E. Stinson and J. C. Mitchell, "Characterizing bots' remote control behavior," in *Proc. Int. Conf. Detection Intrusions Malware, Vulnerability Assessment*, 2007, pp. 89–108.

[61] Y. Xue, Z. Xu, M. Chandramohan, and Y. Liu, "Accurate and scalable cross-architecture cross-OS binary code search with emulation," *IEEE Trans. Softw. Eng.*, vol. 45, no. 11, pp. 1125–1149, Nov. 2019.

[62] M. Fan *et al.*, "Android malware familial classification and representative sample selection via frequent subgraph analysis," *IEEE Trans. Inf. Forensics Secur.*, vol. 13, no. 8, pp. 1890–1905, Aug. 2018.

[63] W. Zhang, H. Wang, H. He, and P. Liu, "DAMBA: Detecting android malware by ORGB analysis," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 55–69, Mar. 2020.

[64] W. Wang, J. Wei, S. Zhang, and X. Luo, "LSCDroid: Malware detection based on local sensitive API invocation sequences," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 174–187, Mar. 2020.

[65] Z. Wu, S. Gianvecchio, M. Xie, and H. Wang, "Mimimorphism: A new approach to binary code obfuscation," in *Proc. 17th ACM Conf. Comput. Commun. Secur.*, 2010, pp. 536–546.

[66] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proc. 23rd Annu. Comput. Secur. Appl. Conf.*, 2007, pp. 421–430.

[67] J. Ming, D. Xu, L. Wang, and D. Wu, "LOOP: Logic-oriented opaque predicate detection in obfuscated binary code," in *Proc. 22nd ACM SIGSAC Conf. Comput. Commun. Secur.*, 2015, pp. 757–768.

[68] M. Fredrikson, S. Jha, M. Christodorescu, R. Sailer, and X. Yan, "Synthesizing near-optimal malware specifications from suspicious behaviors," in *Proc. IEEE Symp. Secur. Privacy*, 2010, pp. 45–60.

[69] G. Jacob, R. Hund, C. Kruegel, and T. Holz, "JACKSTRAWS: Picking command and control connections from bot traffic," in *Proc. 20th USENIX Conf. Secur.*, San Francisco, CA, USA, 2011, pp. 1–29.

[70] H. Lu, X. Wang, B. Zhao, F. Wang, and J. Su, "ENDMal: An anti-obfuscation and collaborative malware detection system using syscall sequences," *Math. Comput. Model.*, vol. 58, no. 5, pp. 1140–1154, 2013.

[71] Z. Tian, T. Liu, Q. Zheng, E. Zhuang, M. Fan, and Z. Yang, "Reviving sequential program birthmarking for multithreaded software plagiarism detection," *IEEE Trans. Softw. Eng.*, vol. 44, no. 5, pp. 491–511, May 2018.

[72] D. Basin, S. Mödersheim, and L. Vigano, "An on-the-fly model-checker for security protocol analysis," in *Proc. Eur. Symp. Res. Comput. Secur.*, 2003, pp. 253–270.

[73] H. Wang, T. Liu, X. Guan, C. Shen, Q. Zheng, and Z. Yang, "Dependence guided symbolic execution," *IEEE Trans. Softw. Eng.*, vol. 43, no. 3, pp. 252–271, Mar. 2017.

[74] J. Clause, W. Li, and A. Orso, "Dytan: A generic dynamic taint analysis framework," in *Proc. Int. Symp. Softw. Testing Anal.*, 2007, pp. 196–206.

[75] S. K. Nair, P. N. Simpson, B. Crispo, and A. S. Tanenbaum, "A virtual machine based information flow control system for policy enforcement," *Electron. Notes Theor. Comput. Sci.*, vol. 197, no. 1, pp. 3–16, 2008.

[76] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proc. 14th ACM Conf. Comput. Commun. Secur.*, 2007, pp. 116–127.

[77] F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su, "X-force: Force-executing binary programs for security applications," in *Proc. USENIX Secur. Symp.*, 2014, pp. 829–844.

[78] T. Barabosch, S. Eschweiler, and E. Gerhards-Padilla, "Bee master: Detecting host-based code injection attacks," in *Proc. Int. Conf. Detection Intrusions, Vulnerability Assessment*, 2014, pp. 235–254.

[79] G. S. Kc, A. D. Keromytis, and V. Prevelakis, "Countering code-injection attacks with instruction-set randomization," in *Proc. 10th ACM Conf. Comput. Commun. Secur.*, 2003, pp. 272–280.

[80] Y. Xue *et al.*, "Auditing anti-malware tools by evolving android malware and dynamic loading technique," *IEEE Trans. Inf. Forensics Secur.*, vol. 12, no. 7, pp. 1529–1544, Jul. 2017.

[81] K. Qiu, Z. Zheng, K. S. Trivedi, and B. Yin, "Stress testing with influencing factors to accelerate data race software failures," *IEEE Trans. Rel.*, vol. 69, no. 1, pp. 3–21, Mar. 2020.