From the "Click Trajectories" paper:

1. What is the motivation/goal(s) of this work?

   To study the structural dependencies—and hence the potential weaknesses—within the spam ecosystem's business processes, so as to guide decisions about the most effective mechanisms for addressing the spam problem.

2. What are the necessary infrastructure to host a spam website (i.e., the "click support")?

   Redirection sites, domains, name servers, web servers, stores and affiliate programs.

3. What are the three strategies to disrupt the spammer's business model?

   a. Blocking its advertising (e.g., filtering spam).
   b. Disrupting its click support (e.g., takedowns for name servers of hosting sites).
   c. Interfering with the realization step (e.g., shutting down merchant accounts).

4. What are the findings of this study (i.e., have they achieved their research goals)?

   The payment system is the weakest link of the spam infrastructure: it's highly concentrated and the cost of switching is high. The spam ecosystem can be disrupted effectively if U.S. banks refuses to settle transactions with the banks identified as supporting spam-advertised goods.

From the "Effective and Efficient Malware Detection at the End Host" paper:

1. What are nodes and edges in the behavior graph?

   Each node/vertex represent a system call. An edge between syscall `x` and `y` represents a data dependency between `x`'s output and `y`'s input.

2. Why system calls and why using a graph instead of sequences?

   Most malicious behaviors requires system calls. Sequence-based signatures may capture superficial (temporal) dependencies thus can be bypassed by reordering syscalls. Graph-based signatures capture the true (data) dependencies between syscalls.

3. The key to achieve "efficient detection" is to avoid using expensive dynamic data flow tracking (tainting) to determine the dependencies between system calls (i.e., match the edges during runtime), how does the proposed method work?

At analysis time, a program slice that is responsible for reading the input and transforming it into the corresponding output is extracted from the observed data flow. Then a symbolic expression is derived to represent the semantics of the slice. At testing/matching time, the observed inputs are used to calculate the expected output, and match with the observed output. If matches, a data-flow (dependency) is detected.

From the "Virtual Machine Introspection" paper:

1. Why the authors propose a virtual machine introspection (VMI)-based approach (i.e., what are their motivations)?

   Network-based intrusion detection systems (NIDS) offer high attack resistance but low visibility (i.e., can be evaded), and host-based intrusion detection systems (HIDS) offer high visibility but sacrifice attack resistance (i.e., can be attacked). VMI-based IDS offer both good visibility into the state of the monitored host and strong isolation, thus lending significant resistance to both evasion and attack.

2. What is the main challenge in developing a VMI-base host intrusion detection system (HIDS)?

   An IDS running outside of a virtual machine only has access to hardware-level state (e.g. physical memory pages and registers) and events (e.g. interrupts and memory accesses), instead of OS level abstractions, which are commonly used to specify IDS policies.

3. Can a VMI-based HIDS be attacked? If so, name two possible methods.

   Yes. Section 8 lists several possible attack methods. They can be generally categorized into: attacking the VMM and attacking the IDS.

From the "Smashing the Stack for Fun and Profit" paper:

1. What is NOP sledge and why it is useful?

   NOP sledge is a sequence of NOP instructions before the real shellcode. It is used to increase the robustness of the exploit: even if the guessed address of the shellcode is off by some bytes, as long as the hijacked control flow hit the NOP sledge, it will lead to a successful execution of the shellcode.

2. What is the simplest way to find potential buffer overflow vulnerabilities?

   Searching the source code (e.g., with `grep`) to find invocation of dangerous C library APIs (e.g., `strcat()`, `strcpy()`, `sprintf()`, and `vsprintf()`).

From the "StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks" paper:

1. Why did it mitigate the easy-to-guess canary problem?

If the attacker can easily guess the canary value, then the attack string can include the canary word in the correct place and bypass the protection. To deal with easily-guessed canaries, StackGuard chooses a set of random canary words at the time the program starts.

2. MemGuard provides better protection than StackGuard, why it is not widely adopted like StackGuard?

   Because of the performance overhead: the cost of a write to a non-protected word on a protected page is approximately 1800 times the cost of an ordinary write.

From the "The Geometry of Innocent Flesh on the Bone: Return-into-libc without Function Calls (on the x86)" paper:

1. Why does the Galileo algorithm scans backwards instead of forwards?

   Because it is simpler: when scanning backwards, the sequence-so-far forms the suffix for all the sequences we discover, so the sequences will all start at instances of the `ret` instruction.

2. Explain how the found shellcode works.

   Unique to each machine. High-level template:

   a. Prepare `"/bin/sh"` somewhere, and followed by a `NULL` word.
   b. Load the addresses into syscall parameters.
   c. Prepare the syscall number.
   d. Invoke the system call.

From the "Control-Flow Integrity Principles, Implementations, and Applications" paper:

1. What are the three requirements/assumptions for CFI enforcement?

   - Unique IDs: After CFI instrumentation, the bit patterns chosen as IDs must not be present anywhere in the code memory except in IDs and ID-checks.
   - Non-writable Code: It must not be possible for the program to modify code memory at runtime.
   - Non-executable Data: It must not be possible for the program to execute data as if it were code.

2. What is the strategy mentioned in the paper that can improve the accuracy of CFG?

   Code duplication: code duplication can be used for eliminating the possibility of overlapping but different destination sets. For instance, two separate copies of the function `strcpy` can target two different destination sets when they return.

From the “An empirical study of the reliability of UNIX utilities” paper:

1. What experience motivate the authors to conduct this more systematic experiment? Please explain what was the source of random inputs in that experience.

   On a dark and stormy night one of the authors was logged on his workstation on a dial-up line from home. The rain (as the random source) caused frequent spurious characters on the line. And these random spurious characters were causing programs to crash.

2. What is the most common cause for crash?

   Spatial memory errors (e.g., out-of-bound access) is the most common cause for crash.

From the “EXE: Automatically Generating Inputs of Death” paper:

1. How does EXE (or what does exe-cc insert to) find inputs that can crash the program?

   `exe-cc` inserts code that calls to check if a symbolic expression could have any possible value that could cause either (i) a null or out-of-bounds memory reference or (ii) a division or modulo by zero.

2. How does EXE map C code to symbolic constraints?

   EXE maintains a table that maps addresses of bytes/arrays to their corresponding symbolic expressions.

   - $v := e$: EXE builds the symbolic expression e_{sym} representing e, and records that $\&v$ maps to it.
   - $b[e]$: EXE allocates an identically-sized STP array b_{sym}, initializes it to have the same (constant) contents as b, and records that b maps to b_{sym}.

From the “A First Step Towards Automated Detection of Buffer Overrun Vulnerabilities” paper:

1. In the “Smashing the Stack for Fun and Profit” paper, the author suggested using grep to find use of dangerous libc APIs like strcat as potential location for buffer overflow.

   Compare to that simple static analysis (i.e., `grep strcat`), how does the approach proposed in this paper improves the precision and reduces false positives? A call to these APIs does not always lead to buffer overflow. The approach proposed in this paper improves the precision by

(i) tracking the the number of bytes allocated for the string buffer (its allocated size), and the number of bytes currently in use (its length); and

(ii) checking, for each string buffer, whether its inferred allocated size is at least as large as its inferred maximum length.

2. Static analysis trades precision for scalability, give an example of the imprecise modelings (i.e., heuristics) discussed in this paper.

   - The analysis is flow-insensitive, i.e., ignore all control flow and disregard the order of statements.
   - The analysis does not handle pointers operations and doubly-indirected pointers (e.g., `char **`).
   - The analysis ignores all function pointers.

From the "Android Permissions Remystified: A Field Study on Contextual Integrity" paper:

1. What is "contextual integrity?"

   Personal information should be used in ways the match users' (plural) expectations.

2. What kind of method have the authors tried to reduce the frequency of runtime permission requests?

   The authors constructed several statistical (machine learning) models to predict users' desire to block certain permission requests using the contextual data.

From the "Preventing Privilege Escalation" paper:

1. When performing "privilege separation," what principle should be followed? In other words, if we treat this as an optimization problem, what should be maximized/minimized under what constraints?

   Minimize the amount of code that runs with special privilege, without affecting or limiting the functionality of the service.

2. What kind of privileges are required by the "unprivileged child?"

   - Talk the the remote client over the network
   - Send requests to the "monitor" via IPC

From the "Native Client: A Sandbox for Portable, Untrusted x86 Native Code" paper:

1. How does native client enforce data integrity (i.e., constrain data access for code inside the inner sandbox)?

On x86 (32-bit), this is achieved by using segmented memory to constrain both data and instruction memory references. On other platforms, including x86-64, check this paper: [Adapting Software Fault Isolation to Contemporary CPU Architectures] (https://www.usenix.org/event/sec10/tech/full_papers/Sehr.pdf).

2. How does native client enforce control-flow integrity?

   Statically, NaCl requires the binary to satisfy

   - C5. The binary contains no instructions or pseudo-instructions overlapping a 32-byte boundary
   - C6. All valid instruction addresses are reachable by a fall-through disassembly that starts at the load (base) address.
   - C7. All direct control transfers target valid instructions. Dynamically, NaCl (i) constraints control transfer to be within the inner sandbox by using the x86 segment register (`CS`), and (ii) applies a mask (`0xffffffe0`) to ensure the target is aligned to 32-byte (i.e., C5).

From the "Kerberos: An Authentication Service for Open Network Systems" paper:

1. What is the threat model (i.e., what attackers can do and what kind of attacks this work aims to mitigate)?

   Attackers capability:

   - Attackers have arbitrary control of the client (workstation).
   - Attackers can be masquerading as a known network server.
   - Attackers can watch the network traffic (e.g., man-in-the-middle capability).

   Design goals:

   - G1: secure, attackers cannot impersonate a client or a server.
   - G2: robust, no single point of failure.
   - G3: transparent, user should no be aware.
   - G4: scalable, not becoming a bottleneck.

2. Authentication can be done based on what you know (e.g., password) and what you have (e.g., a secret), explain how a server authenticate the client, and how the client authenticate the server.

   a. The client is authenticated when getting the initial ticket $T_{c,tgs}$, which is encrypted using the client secret key K_c.

   ```
   Client ---            c, tgs                --> Kerberos
   Client <-- {K_{c,tgs}, {T_{c,tgs}}K_tgs}K_c --- Kerberos
   ```

Because K_c is derived from the user's password, without known the password, an attacker cannot get the ticket and the session key $K_{c,tgs}$. Note that the ticket is encrypted with the ticket generation server's (TGS) secret key K_{tgs}, which is only known to Kerberos and the TGS, so the client cannot forge a ticket.

b. To access a server, the client first requests a ticket from the ticket server

```
Client --- s, {T_{c,tgs}}K_tgs, {A_c}K_{c,tgs} --> TGS
Client <--  {{T_{c,s}}K_s, K_{c,s}}K_{c,tgs}   --- TGS
```

The TGS authenticate the request as follows:

(i) decrypts the ticket $\{s, c, addr, timestamp, life, K_{c,s}\}$ using TGS's secret key K_{tgs},
(ii) extracts the session key $K_{c,tgs}$ from the ticket,
(iii) decrypts the authenticator $\{c, addr, timestamp\}$ using the session key,
(iv) checks if all of the following pairs match: `{s,tgs}`, `{c of the ticket, c of the authenticator}`, `{addr, requesting client's address}`,
(v) checks if the timestamp of the authenticator is current, and the ticket has not expired.

The client is authenticated by (i) having a valid ticket $\{T_{c,tgs}\}K_{tgs}$ and (ii) knowing the session key $K_{c,tgs}$. An attacker monitoring the network can know $\{T_{c,tgs}\}K_{tgs}$ but cannot know $K_{c,tgs}$. A forged authenticator (not encrypted by $K_{c,tgs}$), once decrypted (with $K_{c,tgs}$), will have mismatching information. An unauthorized client cannot forge a ticket, otherwise the session key would be wrong, then information from the (wrongly) decrypted authenticator would not match information from the (forged) ticket.

c. To request a service, the client sends the ticket retrieved in step b from the TGS, and the authenticator (a new and different one from b):

```
Client --- {T_{c,s}}K_s, {A_c}K_{c,s} --> Server
Client <--    {timestamp + 1}K_{c,s}  --- Server
```

The server authenticate the client similar to how TGS authenticate the client: (i) having a valid ticket (only TGS can generate a valid ticket, so TGS should have authenticated the client and authorized the access) and (ii) knowing the session key $K_{c,s}$.

The client authenticate the server by checking the encrypted $\{timestamp + 1\}$: only the real server that knows the secret key K_s can decrypt the ticket and extract the correct session key.

From the “VC3: Trustworthy Data Analytics in the Cloud” paper:

1. Section 5 describes a technique called “self-integrity.” What kind of security threats does self-integrity aims to mitigate?

   Because an enclave program can access outside memory (similar to OS kernel can access user space memory), self-integrity prevents the enclave program from dereferencing pointers to untrusted memory outside the enclave. Writing with a corrupted pointer can leak information, and reading data with a corrupted pointer can lead to control-flow hijacking.

2. Section 6.2 describes the key exchange protocol of VC3, where a secret key `k_w` is exchanged. Explain why only the client and the enclave knows `k_w`, and the untrusted cloud provider (including administrators, OS, and the hypervisor) cannot know `k_w`.

   a. `k_w` is sending to the user after being encrypted with the user’s public key `pk_u`. Because only the user knows the corresponding secret key, the untrusted cloud provider cannot decrypt the message and extract the key.
   b. The encrypted key `m_w` is also included as part of the quote (remote attestation). A valid quote implies that `k_w` is indeed generated inside a SGX enclave with the expected context. This guarantees that the untrusted cloud provider also cannot extract the key from the machine.