I am interested in solving **system security** problems via **program analysis** techniques. More specifically, I have developed fundamental primitives for the investigation of advanced cyber-attacks and the analysis and prevention of ever-evolving malicious programs and payloads across multiple platforms. In particular, my research focuses on building systems to solve three prominent problems in system security: ① attack provenance and root-cause analysis via *causality inference* [1, 2], ② reverse-engineering through *cross-platform binary analysis* [3, 4, 10], and ③ software exploit prevention via *input randomization* [5, 7]. My research has introduced novel techniques along with practical systems, advancing the status quo. In recognition of my contributions, I have been honored with four prestigious awards: *Best Paper Award*, *ACM SIGSOFT Distinguished Paper Award*, *Maurice H. Halstead Memorial Award*, and *Microsoft Most Valuable Professional Award*.

1 Research Overview and Highlights

① **Attack provenance via causality inference.** Recent cyber-attacks are becoming increasingly targeted and sophisticated. A new class of attack called Advanced Persistent Threat, or APT, targets a specific organization and compromises systems over a long period of time (e.g., from months to years) without being detected. For example, the notorious Stuxnet compromised more than hundreds of thousands of systems through multiple steps. It lurked in the systems for years while silently updating, installing backdoors, and exfiltrating information. In the end, it even caused physical damages to thousands of machines. It was commented that the attack could have caused a nuclear disaster more catastrophic than *Chernobyl*. Investigating such an attack is challenging: (1) it occurs over an extended period of time, and hence, the log files for the investigation are prohibitively large (e.g., on the scale of TBs), and (2) the attack process is highly sophisticated involving a large number of processes and files.

To address the above challenges, my research proposed *causality inference* to determine dependencies between system calls (e.g., between input and output system calls) and allow investigators to determine the origin of an attack (e.g., receiving a spam email) and assess the consequences of the attack. I have designed a practical causality inference engine LDX [1] that is *4 times more accurate and 2 orders of magnitude faster* (6% runtime overhead) than state-of-the-art taint analysis techniques. Expanding beyond LDX, I have proposed a model based causality inference system, MCI [2]. MCI is practical as it does not require any modification or instrumentation to end-user systems, and it is more accurate and precise (0.1% FP/FN) than the previous state-of-the-art technique BEEP [13] which does require instrumentation (12.8% FP/0.3% FN). LDX and MCI are currently going through tech-transfer as part of the *DARPA Transparent Computing (TC) Program*, a $5.3 million joint program between SRI International, Purdue Univ., Univ. of Wisconsin-Madison, and Univ. of Georgia.

② **Reverse-engineering through cross-platform binary analysis.** The recent rise of IoT devices (predicted to reach 20.4 billion by 2020 [14]) such as wearable devices, drones, and self-driving cars raises alarming security concerns. In 2016, 7 massive Distributed Denial of Service (DDoS) attacks were all carried out by compromised IoT devices. Popular websites such as Amazon, CNN, and PayPal were affected. In particular, Dyn, a DNS service provider and a victim of one of the attacks, lost 24% of customers. During the attacks, compromised devices coordinated malicious activities through Command and Control (C&C) protocols. Hence, understanding C&C messages is essential for revealing an attacker's intentions. However, there are two prominent challenges: (1) they target various platforms and architectures (such as ARM, MIPS, and AVR) which many advanced analysis tools do not support, and (2) C&C servers are often not accessible in practice, making reverse-engineering of C&C messages difficult.

My research has introduced new paradigms in cross-platform attack investigation (e.g., IoT attack investigation). Rather than re-implementing analysis tools on multiple platforms, my work enables *transforming* a program execution on a platform (e.g., ARM Raspberry Pi) into another platform (e.g., x86 Linux) where various analysis tools (e.g., Pin/Valgrind) are already available. My technique, PIEtrace [3], traces and transforms an IoT program execution into a platform-independent C program that can be compiled and run on other platforms. PIEtrace was honored with the **Best Paper Award** (1/317) and the **ACM SIGSOFT Distinguished Paper Award** (3/317). Moreover, I have developed a novel system, called P2C [4], which reverse-engineers types and semantics of unknown files and C&C messages in the following scenario: *understanding C&C messages generated in the past* and *in the absence of C&C servers*. P2C has been used to reveal the semantics of C&C messages generated by the *Zeus* malware, one of the largest known botnets, which infected *3.6 million systems*.

③ **Software Exploit Prevention via Input Randomization.** Proactively protecting systems against a wide spectrum of software exploit attacks is of the utmost importance. Unfortunately, the ever-expanding variety of attack vectors and methods makes such defenses difficult. Protection targeting specific attacks such as ROP, use-after-free, and type-confusion can hardly keep up with rapidly growing new attack vectors.

I have designed a novel software protection system, A2C [5], that prevents a wide range of attacks by randomizing inputs such that *any malicious payloads contained in the inputs are corrupted*. The protection provided by A2C is both general (e.g., against various attack vectors including buffer-overflow, integer-overflow, use-after-free, type-confusion, and ROP) and practical (7% runtime overhead). More importantly, the idea is applicable to many platforms. Specifically, my colleagues and I have designed PAD [7], a system that prevents malicious payloads in malvertising attacks on web systems via input randomization. PAD successfully prevented *real-world malvertising attacks* including the AdGholas malvertising campaign [15] which affected *thousands of victims everyday for over a year* using a sophisticated steganography technique.

2 Research Projects

2.1 Attack Provenance via Causality Inference

Prior to my work, the two most widely used state-of-the-art techniques for attack investigation were taint analysis and audit-logging. Taint analysis tracks program dependencies by monitoring the data propagation of individual instructions. Audit-logging focuses on dependencies between syscalls exposed through explicit syscall arguments (e.g., file handles). For example, they consider syscalls on the same file or within the same process causally related. Unfortunately, taint analysis suffers from significant performance overhead as it needs to monitor every instruction. Moreover, both taint analysis and audit-logging are inaccurate as taint analysis has difficulty handling control dependencies and the assumptions made in audit-logging (e.g., all output syscalls are causally related to all input syscalls within a process) are too coarse-grained, leading to a large number of bogus dependencies.

Counter-factual causality, first introduced in the 18^{th} century by David Hume [16], can be used to describe the desired causal analysis in an attack investigation. Specifically, given two events, *a latter event is causally dependent on a preceding event if changes at the preceding event lead to state differences in the latter event.* To this end, I realized that the limitations of taint analysis and audit-logging stem from their *imprecise approximations* of counter-factual causality. My research pioneered building techniques that implement precise counter-factual causality for cyber attack investigation.

Conducting faithful counter-factual causality in the context of program and program execution.

My research takes a fundamental approach: *adapting the original counter-factual causality concept in the context of program and program execution.* LDX [1] conducts faithful counter-factual causality inference on computer systems via *dual execution.* Specifically, it runs two executions in parallel — the original execution and its mutated version with mutations on input syscalls. Then, it observes differences at output syscalls. Any difference indicates causality between the mutated input syscalls and the output syscalls. Due to the mutation LDX introduces, the mutated execution may take a different path, leading to a different sequence of executed syscalls, when compared with the original execution. Hence, a fundamental challenge of LDX is to align the two executions so that they can be compared at the same execution point, because comparing executions at misaligned points leads to incorrect causality (i.e., FP/FN). To this end, I designed a novel runtime counter derived from program structure. The counter is not a simple logic timestamp, but rather denotes execution points by ensuring an important key property: The counter value indicates the relative progress of executions, meaning that an execution with a larger counter value must be ahead of another execution with a smaller counter value with respect to program structure. The counter facilitates alignment of two executions, enabling precise and efficient causality inference. Evaluation on a large set of real-world applications, including *Apache web server*, shows that LDX is *4 times more accurate* and *2 orders of magnitude faster* than state-of-the-art taint analysis techniques.

Model based causality inference in audit-logging for practical attack provenance.

The primary hindrance of existing techniques, including LDX, for attack provenance is their requirement of changing end-user systems such as program instrumentation and kernel modification. In contrast, existing automata-based techniques do not require instrumentation. They work by identifying program behaviors (e.g., file downloading) from a concrete log (e.g., a syscall log). They construct automata that represent the behaviors. Then, they parse a log generated from an execution with the automata to determine whether the behaviors are exhibited in the log. However, they do not take dependencies into account; for instance, they may detect two behaviors that are "download a file" and "send a message," while the causal relationship between these two behaviors is not exposed.

MCI [2] is a model-based causality inference technique for attack provenance that directly works on syscall logs *without requiring any end-user program instrumentation or kernel modification.* For each program, it uses LDX to acquire precise *causal models* for a set of primitive operations (e.g., opening a file). A causal model is a sequence of syscalls annotated with *inter-dependencies (causality) between the syscalls within the model*, where some of the inter-dependencies are caused by memory operations and hence implicit at the syscall level. During deployment, MCI parses the existing audit-logs into concrete model instances to derive causality. To this end, parsing syscall logs with causal models with implicit dependency information leads to two prominent challenges: *language complexity* and *ambiguity.* First, to express complex inter-dependencies annotated in causal models, expressive grammar is required while more expressive grammar describes more complex language (e.g., context-free or context-sensitive) and hence leads to higher cost in parsing. Second, some syscalls can be parsed by multiple models that share common parts (e.g., common prefixes). In such cases, it is difficult to decide which model is the right one. As different causalities are derived from different models, the ambiguity problem may lead to incorrect causality (i.e., FP/FN). To solve these challenges, I designed a novel model parsing algorithm called *segmented parsing* that can handle multiple model complexity levels (e.g., regular, context-free, and context-sensitive) and substantially mitigate the ambiguity problem by leveraging *explicit dependencies* that can be directly derived from the log (e.g., dependencies caused by file handles). Specifically, MCI first obtains a *model skeleton* of each causal model. A model skeleton consists of syscalls with explicit dependencies. The skeleton partitions a model into *model segments* that can be described and parsed by automata. Without requiring any changes to end-user systems, MCI recovers causality with close

to 0% FP/FN for most applications (the worst case: 8.3% FP and 5.2% FN). More importantly, causal models have *composability* such that models for primitive operations can be composed together to describe complex system-wide attack behaviors. For example, primitive models for "Edit", "Copy", "Paste", and "Save" can compose a new model that represents a complex user behavior "Edit→Copy→Edit→Paste→Edit→Save" (e.g., potential information exfiltration). Evaluation on *attack cases created by security professionals in the DARPA TC program* shows that attack causal graphs generated by MCI are more precise than those generated by the previous state-of-the-art system BEEP [13] that requires instrumentation.

2.2 Cross-platform Binary Analysis and Data file-format Reverse-engineering.

Due to the prevalence of IoT devices, testing, debugging, and security analysis on IoT devices is becoming increasingly important. However, many advanced debugging and analysis tools do not support new IoT platforms. Not surprisingly, an HP study [17] revealed 70% of IoT devices contain vulnerabilities. Notably, many of these vulnerabilities are not new, hence if existing analysis tools were applicable on IoT devices, then these vulnerabilities could have been detected and patched before deployment. My research takes a unique approach: rather than porting existing analysis tools to IoT platforms, I propose transforming a *platform-dependent program execution* (e.g., an IoT platform's execution) into a *platform-independent program* that can be executed on other preferred platforms so that *existing analysis tools* can be leveraged.

Cross-platform binary analysis via platform-independent trace program.

PIEtrace [3] virtualizes a program execution on an IoT device and transforms *a trace of the execution* into a *platform-independent C program* that can be compiled and executed on other platforms, which we call *a trace program*. The trace program reproduces the original execution including control and data dependencies, hence debugging/analyzing the trace program is equivalent to debugging/analyzing the original execution on the IoT device. This approach faces two prominent challenges. First, instructions are platform-dependent. Second, syscalls and their return values are also platform-dependent. To eliminate such dependencies, PIEtrace first virtualizes platform-dependent instructions. For syscalls and their return values, I developed *a lazy-logging algorithm* which captures all platform-dependent memory updates such as those made by syscalls. Intuitively, PIEtrace takes snapshots of the memory before and after every syscall. After a syscall, it compares the two snapshots to identify whether the memory was updated by the syscall. If the two snapshots are different, it means the syscall has side effects. In such case, PIEtrace replaces the syscall instruction with concrete values from the snapshot taken after the syscall. The algorithm does not require understanding the underlying syscall interfaces. PIEtrace has facilitated identifying/debugging sensors and IoT devices vulnerabilities on various platforms using existing program analysis tools such as Pin and Valgrind on Windows/Linux. Due to the novelty of the idea, the PIEtrace publication received the *ASE Best Paper Award* (a top-tier conference in software engineering) and the *ACM SIGSOFT Distinguished Paper Award*.

Another prominent challenge facing IoT devices is a lack of software support. Re-implementing software is time-consuming and error-prone (often introducing new bugs or vulnerabilities). Again, my research proposes a different approach: *extracting components from existing binary programs* and *reusing* the extracted components on *new platforms*. CPR [6] leverages PIEtrace to extract desired components from a binary application and create a standalone platform-independent program. Specifically, CPR merges multiple trace programs generated by PIEtrace to synthesize a program which can take *new inputs* (other than the inputs used to generate the trace programs) and produce *outputs corresponding to the inputs*. Conversely, a trace program generated by PIEtrace *does not take any input and only regenerates the traced execution*. CPR successfully extracts and reuses components from Windows binaries (without source code and symbolic information) on *Linux*, *Raspberry PI*, and *Cisco IOS*.

Understanding unknown file-format by monitoring execution.

Malicious programs often create suspicious files or network messages. As they often contain information that attackers want to exfiltrate, understanding these files/messages is essential for attack investigation. However, this is challenging as the file/message format is unknown. To this end, I developed P2C [4] to reverse-engineer unknown file formats, including field types and semantics, by observing program execution. Specifically, given an unknown file, P2C recovers the file-format by analyzing how the program accesses the file and how the file content is used in the program. For example, if there is a loop that reads certain parts of a file, the parts likely belong to the same buffer. Furthermore, if a buffer is passed to a file open API, the buffer likely contains a file name. Unfortunately, in practice, investigators rarely can obtain a program that can parse and understand an unknown file (called *a consumer program*) for P2C to analyze. For instance, a botnet consists of a number of malicious clients (called *bots*) on victim machines and C&C servers on the attackers' machines. Bots create and send C&C messages to the servers. In this context, the bots are *producers* as they produce the messages, and the servers are *consumers* as they can parse and understand the messages from the bots. Unfortunately, a C&C server (a consumer) is often not available on the victim's system, hence cannot be analyzed. P2C solves this challenge by leveraging the following key observation: *producers and consumers have symmetric program structure*. For example, a producer (e.g., the bot client) may construct a simple network message by first *writing a size* of some data and then *appending the data*. Symmetrically, the corresponding consumer (e.g., the C&C server) reads the message by first *reading the size* and then *reading the data according to the size*. Leveraging this symmetry, P2C recovers unknown file-format without a consumer by transforming a

producer program execution into a consumer program execution. Moreover, in some cases, a user may not even know the exact producer of an unknown file/message. The user may only know a set of potential producers. P2C can identify the true producer from the set of producers by leveraging the observation that *if a producer candidate is not the true producer, then the transformation fails early in its execution.* P2C is capable of revealing the semantics of C&C messages generated by one of the largest botnets: *Zeus*. Moreover, P2C can recover secret text containing exfiltrated credit card numbers hidden via *steganography* in a seemingly benign image.

2.3 Attack Prevention via Input Randomization

To build general protection against ever-evolving cyber attacks, my research breaks a common critical step of most attacks: *malicious payload injection and execution.* In particular, I exploit a fundamental characteristic of malicious payloads: they are designed with strict semantic assumptions about the execution environment (e.g., platform or architecture), hence they are *particularly brittle* to any mutation.

Corrupting malicious payloads via input mutation

A2C [5] exploits *the brittleness* of malicious payloads to provide general protection. It corrupts malicious payloads by encoding all inputs from untrusted sources at runtime. However, the encoding may break program execution on benign inputs as well. To assure that the program continues to function correctly when benign inputs are provided, I developed a static analysis technique that identifies all the places that read and process inputs and selectively inserts decoding logic at some of those places. Specifically, decoding only occurs when the use of the inputs cannot be exploited. For instance, when inputs in a byte array are copied to an integer array, each byte of the inputs is padded with 3 zero bytes (as an integer is 4 bytes on 32-bit machines) before it is stored into the integer array. Constructing a meaningful payload with 3 zero bytes in every four bytes is extremely difficult, if not impossible. To this end, I proposed a novel *constraint solving algorithm* which identifies operations that make inputs no longer exploitable, such as the copy operation from a byte array to an integer array. The operations essentially divide the state space of a program into *exploitable* and *post-exploitable* sub-spaces because the program state before the operation is exploitable, but no longer so after the operations. Therefore, A2C decodes the mutated values only when they are transmitted from the exploitable space to the post-exploitable space. Notably, the exploitable space is much smaller than the post-exploitable space — making A2C highly efficient. A2C successfully achieves general protection for a large set of real-world programs, including *Apache web server* against a variety of attacks (e.g., heap spraying, use-after-free, buffer-overflow, integer-overflow, and type-confusion) with low overhead (6.94%). PAD [7] was also inspired by the same methodology of breaking malicious payloads via input mutation to prevent malvertising which is among the largest online attack campaigns, targeting *millions of users everyday*. PAD successfully prevented real-world malvertising attacks proving that the idea is applicable to other platforms.

2.4 Other Projects

I have successful research collaborations on web security [11, 12], ARM binary (e.g., commercial drones' firmware binaries) rewriting [10], and online education [8, 9].

3 Future Research

In the future, I am enthusiastic about continuing to integrate practical attack provenance and advanced binary analysis. Specifically, I plan to focus on *causality inference for ubiquitous computing environments*, where more than 8.4 billion diverse devices are interconnected. More specifically, computing systems in the interconnected cyber-world act as black boxes hence provide little to no visibility of their internal workings, making attack provenance and investigation challenging. I envision making the interconnected cyber-world computation more *transparent* (when needed), by detecting and tracking causality/provenance across various computational units. Among them, I am particularly interested in enabling precise causality inference on three fundamental components: Ⓐ emerging IoT devices, Ⓑ the wide-spread virtualized environment, and Ⓒ the brain of systems: Artificial Intelligence based systems.

Ⓐ IoT Attack Provenance. In analyzing IoT systems, my research will overcome the following prominent challenges:

Heterogeneous execution models. Many IoT systems have a unique program execution model which is highly asynchronous, preemptive, and event-driven. As such, their execution is dominated by event handling where an event may be preempted by another (more critical event). This execution model makes causality inference and provenance tracking particularly challenging. My prior work, MCI [2], deals with exception handlers and thread-interleavings that have similar nature to the preemptive IoT execution model, using a method similar to Push-down Automata. I envision extending the technique to handle the IoT execution model challenge.

Cyber-Physical Awareness. IoT devices employ integral physical components that interact with their physical surroundings. Traditional program analysis techniques focusing on software systems hence have difficulty tracking causality, especially when dependencies have to go through the physical component. For instance, consider two variables in a program, x and y, where x is used to compute the motor throttle of a quad-rotor and y is computed from the velocity sensor reading.

While causality between the two variables is apparent, existing program analysis techniques can hardly capture such causality because it has to go through the physical component. I envision my model-based causality inference can be naturally extended to model causality in the physical world.

Resource Adaptability. Limited resources (e.g., computation power, storage, and battery) on IoT systems make complex runtime analyses unaffordable. For instance, audit-logging is often disabled on IoT devices due to its substantial resource consumption. From my experience, I observe that a lengthy and verbose audit-log can be substantially reduced using a succinct representation. My prior works [5, 6] leverage execution context encoding techniques that can reduce a long sequence of execution trace to an integer value with negligible runtime overhead ($< 2\%$). I plan to extend it to IoT logging.

Ⓑ Causality Inference in Virtualized Environment. Virtualized environments are prevalent and causality inference within such environments is challenging. For instance, most web-browsers support JavaScript (JS) which can implement complex business logic. Given an audit log of a browser that contains numerous syscalls, it is extremely difficult to determine which parts belong to the intrinsic browser logic (e.g., following a link and opening a tab) and which parts are caused by the JS logic of individual webpages (e.g., processing credit card transactions). Therefore, causality inference becomes extremely challenging. In fact, the problem exists in almost all virtualized environments. Considering 37% of network attacks targeting virtualized environments (e.g., browsers) in 2015 [18], understanding causality through the layers of virtualized environments is important. Therefore, I plan to explore *causality introspection* in virtualized environments. In particular, I realize that the root cause of the problem is that execution of all virtualization layers is indistinguishable at the syscall level. Hence, I will explore schemes to partition an execution into smaller units, each belonging to a specific layer, leveraging the execution partitioning analysis technique developed in my research group [13].

Ⓒ Provenance in AI Driven Systems. Recently, Artificial Intelligence (AI) has been adopted to solve complex problems in various domains including facial and voice recognition, self-driving cars, and malware and intrusion detection. Unfortunately, researchers have recently identified various methods to compromise the decision processes of AI based systems. Among them, the *dirty data* problem means that inaccurate, incomplete, erroneous, or even malicious data can lead to inaccurate data analytic results and misguided decision. According to leading industries such as IBM and Twitter, the dirty data problem costs the US economy up to $3.1 trillion a year. Hence, it is critical to track provenance that links a decision outcome to its data sources, especially those that are ill-formatted or malicious. I plan to extend attack provenance tracking in the area of AI oriented data processing by combining program analysis and data analysis.

References

[1] **Y. Kwon**, D. Kim, W. N. Sumner, K. Kim, B. Saltaformaggio, X. Zhang, and D. Xu. LDX: Causality Inference by Lightweight Dual Execution. In *Proc. of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*.

[2] **Y. Kwon**, F. Wang, W. Wang, K. H. Lee, W. Lee, S. Ma, X. Zhang, D. Xu, S. Jha, G. Ciocarlie, A. Gehani, and V. Yegneswaran. MCI: Modeling-based Causality Inference in Audit Logging for Attack Investigation. In *Proc. of the 25th Annual Network and Distributed System Security Symposium (NDSS'18)*.

[3] **Y. Kwon**, X. Zhang, and D. Xu. PIEtrace: Platform Independent Executable Trace. In *Proc. of the 28th IEEE/ACM International Conference on Automated Software Engineering (ASE'13)*.

[4] **Y. Kwon**, F. Peng, D. Kim, K. Kim, X. Zhang, D. Xu, V. Yegneswaran, and J. Qian. P2C: Understanding Output Data Files via On-the-Fly Transformation from Producer to Consumer Executions. In *Proc. of the 22nd Annual Network and Distributed System Security Symposium (NDSS'15)*.

[5] **Y. Kwon**, B. Saltaformaggio, I. Kim, K. H. Lee, X. Zhang, and D. Xu. A2C: Self Destructing Exploit Executions via Input Perturbation. In *Proc. of the 24th Annual Network and Distributed System Security Symposium (NDSS'17)*.

[6] **Y. Kwon**, W. Wang, Y. Zheng, X. Zhang, and D. Xu. CPR: Cross Platform Binary Code Reuse via Platform Independent Trace Program. In *Proc. of the 26th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'17)*.

[7] W. Wang, **Y. Kwon**, Y. Zheng, Y. Aafer, I. Kim, W. Lee, Y. Liu, W. Meng, X. Zhang, and P. Eugster. PAD: Programming Third-party Web Advertisement Censorship. In *Proc. of the 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE'17)*.

[8] D. Kim, **Y. Kwon**, W. N. Sumner, X. Zhang, and D. Xu. Dual Execution for On the Fly Fine Grained Execution Comparison. In *Proc. of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*.

[9] D. Kim, **Y. Kwon**, P. Liu, I. Kim, D. M. Perry, X. Zhang, and G. Rodriguez-Rivera. Apex: Automatic Programming Assignment Error Explanation. In *Proc. of the 27th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'16)*.

[10] T. Kim, C. H. Kim, H. Choi, **Y. Kwon**, B. Saltaformaggio, X. Zhang, and D. Xu. RevARM: A Platform-Agnostic ARM Binary Rewriter for Security Applications. In *Proc. of the 33rd Annual Conference on Computer Security Applications (ACSAC'17)*.

[11] K. Kim, I. Kim, C. H. Kim, **Y. Kwon**, Y. Zheng, X. Zhang, and D. Xu. J-Force: Forced Execution on JavaScript. In *Proc. of the 26th International Conference on World Wide Web (WWW'17)*.

[12] W. Wang, Y. Zheng, X. Xing, **Y. Kwon**, X. Zhang, and P. Eugster. WebRanz: Web Page Randomization For Better Advertisement Delivery and Web-Bot Prevention. In *Proc. of the 24th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE'16)*.

[13] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *Proc. of the 20th Annual Network and Distributed System Security Symposium (NDSS'13)*.

[14] Gartner Says 8.4 Billion Connected "Things" Will Be in Use in 2017, Up 31 Percent From 2016. https://www.gartner.com/newsroom/id/3598917.

[15] Massive AdGholas Malvertising Campaigns Use Steganography and File Whitelisting to Hide in Plain Sight. https://www.proofpoint.com/us/threat-insight/post/massive-adgholas-malvertising-campaigns-use-steganography-and-file-whitelisting-to-hide-in-plain-sight.

[16] D. Hume. An Enquiry concerning Human Understanding, 1748.

[17] HP Study Reveals 70 Percent of Internet of Things Devices Vulnerable to Attack. http://www8.hp.com/us/en/hp-news/press-release.html?id=1744676.

[18] McAfee Labs Quarterly Threat Report March 2016. https://www.mcafee.com/us/resources/reports/rp-quarterly-threats-mar-2016.pdf.