AFRL-RI-RS-TR-2016-070

SPARCHS: SYMBIOTIC, POLYMORPHIC, AUTOMATIC, RESILIENT, CLEAN-SLATE, HOST SECURITY

COLUMBIA UNIVERSITY

MARCH 2016

FINAL TECHNICAL REPORT

APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED

STINFO COPY

AIR FORCE RESEARCH LABORATORY
INFORMATION DIRECTORATE

■ **AIR FORCE MATERIEL COMMAND** ■ **UNITED STATES AIR FORCE** ■ **ROME, NY 13441**

NOTICE AND SIGNATURE PAGE

Using Government drawings, specifications, or other data included in this document for any purpose other than Government procurement does not in any way obligate the U.S. Government. The fact that the Government formulated or supplied the drawings, specifications, or other data does not license the holder or any other person or corporation; or convey any rights or permission to manufacture, use, or sell any patented invention that may relate to them.

This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09. This report is available to the general public, including foreign nationals. Copies may be obtained from the Defense Technical Information Center (DTIC) (http://www.dtic.mil).

AFRL-RI-RS-TR-2016-070 HAS BEEN REVIEWED AND IS APPROVED FOR PUBLICATION IN ACCORDANCE WITH ASSIGNED DISTRIBUTION STATEMENT.

/ S /
TODD CUSHMAN
Work Unit Manager

/ S /
WARREN H. DEBANY, JR.
Technical Advisor, Information
Exploitation & Operations Division
Information Directorate

This report is published in the interest of scientific and technical information exchange, and its publication does not constitute the Government's approval or disapproval of its ideas or findings.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.
PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
MARCH 2016	FINAL TECHNICAL REPORT	SEP 2010 – SEP 2015

4. TITLE AND SUBTITLE

SPARCHS: SYMBIOTIC, POLYMORPHIC, AUTOTOMIC, RESILIENT, CLEAN-SLATE, HOST SECURITY

5a. CONTRACT NUMBER N/A

5b. GRANT NUMBER FA8750-10-2-0253

5c. PROGRAM ELEMENT NUMBER 62303E

6. AUTHOR(S)

Simha Sethumadhavan, Salvatore Stolfo, Angelos D. Keromytis, Junfeng Yang (Columbia University)

David August (Princeton University)

5d. PROJECT NUMBER CRSH

5e. TASK NUMBER CO

5f. WORK UNIT NUMBER LU

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Columbia University
Department of Computer Science M.C. 0401
1214 Amsterdam Avenue
New York, NY 10027-7003

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

Air Force Research Laboratory/RIGA
525 Brooks Road
Rome NY 13441-4505

DARPA/I2O
675 North Randolph St
Arlington, VA 22203-2114

10. SPONSOR/MONITOR'S ACRONYM(S) AFRL/RI

11. SPONSOR/MONITOR'S REPORT NUMBER AFRL-RI-RS-TR-2016-070

12. DISTRIBUTION AVAILABILITY STATEMENT

Approved for Public Release; Distribution Unlimited. This report is the result of contracted fundamental research deemed exempt from public affairs security and policy review in accordance with SAF/AQR memorandum dated 10 Dec 08 and AFRL/CA policy clarification memorandum dated 16 Jan 09.

13. SUPPLEMENTARY NOTES

14. ABSTRACT

The SPARCHS project proposed a new computer systems design methodology that considers defensive security as a first-order design requirement at all levels in computer systems stack.
A defensive mindset to system design means that each component, be it hardware, software or the security mechanism, should be designed assuming that it can be compromised, each component must individually have the ability to detect attacks, limit losses, recover from attacks and learn to prevent future attacks.

15. SUBJECT TERMS

Vulnerability Detection and Mitigation, Static Analysis, Dynamic Confinement, Code Diversification

16. SECURITY CLASSIFICATION OF: a. REPORT	b. ABSTRACT	c. THIS PAGE	17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
U	U	U	UU	44	TODD CUSHMAN
					19b. TELEPHONE NUMBER (*Include area code*) N/A

Standard Form 298 (Rev. 8-98)
Prescribed by ANSI Std. Z39.18

Table of Contents

LIST OF FIGURES II

1.0 SUMMARY 1

2.0 INTRODUCTION 2

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES 3

4.0 RESULTS AND DISCUSSION 3

4.1 HARDWARE SECURITY 3
4.1.1 Securing Hardware Design 4
4.1.2 Measuring and Mitigating Micro-architectural Side Channels 6
4.1.2.1 Spy in the Sandbox: A vulnerability study 6
4.1.2.2 Time Warp 7
4.1.2.3 Side-Channel Vulnerability Factor 9
4.1.3 Architectural Support 11
4.1.3.1 A Hardware Anti-Virus 11
4.1.3.2 Instruction Set Randomization 12
4.1.3.3 Information Flow Tracking 14

4.2 FIRMWARE SECURITY 17
4.2.1 Router Vulnerability Study 17
4.2.2 Printer Vulnerability Study 17
4.2.3 Symbiotic Embedded Machines 18

4.3 OPERATING SYSTEM SECURITY 19
4.3.1 KGuard 19
4.3.2 Ret2dir: Re-thinking Kernel Isolation 20

4.4 APPLICATION SECURITY 20
4.4.1 Information Flow Tracking using Binary Rewriting 20
4.4.2 ROP mitigations 22
4.4.3 Destructive Code Reads in HeisenByte 24

4.5 CONCURRENT SYSTEM SECURITY 25
4.5.1 A Vulnerability Study 25
4.5.2 Solutions 25
4.5.3 Applications 28

4.6 COMPILER OPTIMIZATIONS FOR SECURITY 31
4.6.1 Information Flow Tracking 31
4.6.2 Region Based Memory Safety 31

4.7 MISCELLANEOUS WORKS 32

4.8 TRANSITION EFFORTS 33
4.8.1 AFRL FANCI/CRADA Transition 33
4.8.2 WHCA Symbiote Transition 33
4.8.3 Navy PLC Transition 33

4.9 SYSTEM DEVELOPMENT 33

5.0 CONCLUSIONS 34

6.0 REFERNCES 34

List of Figures

Figure 1: Summary of Contributions.. 1
Figure 2: Stages of Hardware Design.. 5
Figure 3: Algorithm for computing control value.. 5
Figure 4: Algorithm for flagging suspicious wires in the design.. 5
Figure 5: ISR Encryption Process.. 14
Figure 6: DIFT modifications in WHISK.. 16

1.0 SUMMARY

The SPARCHS project proposed a new computer systems design methodology that considers defensive security as a first-order design requirement at all levels in computer systems stack.

A defensive mindset to system design means that each component, be it hardware, software or the security mechanism, should be designed assuming that it can be compromised, each component must individually have the ability to detect attacks, limit losses, recover from attacks and learn to prevent future attacks.

To achieve defensive security, the SPARCHS project drew inspiration from biological organisms that have survived centuries of predatory behavior. Specifically, SPARCHS adapts the following biological primitives to improve security of computer systems: (1) Innate Immunity for detection and isolation of exploits, (2) Diversity for attack prevention, (3) Symbiotic Immunity for protection and detection techniques, (4) Adaptive Immunity for prevention, (5) Optimized redundant execution for continued execution, and (6) Autotomy to contain damage when all else fails.

System Level	Security Technique
Hardware	• Technique for statically checking hardware for backdoors**. Transition effort.
Micro Architecture	• Side channels are a fundamental threat to confidentiality • Becoming a problem in cloud environments and mobile phones. • Techniques for measuring and mitigating side channels*
	• SW AV has same bug density as regular SW, attackers turn off SW AV. • Demonstrated feasibility of building AV in HW (x86 and ARM)*. • World's first Hardware Anti-Virus system.
Architecture	• Instruction Set Randomization: What if each system had a different random, secret ISA? Diversification increases work for the attacker. • Reduced overhead to ~0% Leon Prototype.
	• Hardware Support for DIFT: Many 3rd Party IP components are used in designs today. Not all are designed to support DIFT. • New HW system architecture to allow plug-n-play security.
	• Instruction Set Architecture w/ meta data to check control-flow (Princeton)
Firmware	• Remove vulnerable portions of firmware binaries; automatically create variants • The world's first Host "Anti-virus" for (legacy) embedded systems • Autotomic Binary Structure Randomization. Transition Effort.
OS	• Practical, deterministic thread scheduling systems* • Verifying system rules using symbolic execution
Runtime/ Binary Inst.	• Technique for mitigating ROP attacks* • K Bouncer SW uses existing HW features on x86 (LBR) to detect attacks
	• K Guard – Technique for protecting Ret2usr attacks • Checks control flow integrity between user and kernel code
	• A library for dynamic information flow tracking using PIN
Compiler	• Several techniques to protect against hardware faults (Princeton)
New Security Paradigsm	• Code-lets the continuously monitor program properties aka Symbiotes • Embedded in "gaps" in programs; codelets diversified to confound attackers. • Demo'ed on routers, printers and IP phones. Transition Effort.
	• Recursively fortified Symbiotes • To answer the question "Who monitors the monitor?" • Small prototype with three electrical meters completed.
Vulnerability Studies	• Exploited CISCO IOS Routers, HP Printers, CISCO IP Phone and racy programs • Significant Impact: Engaged with HP/Cisco. Lead to patches. • Attacks have informed (Symbiote) based defenses

Figure 1: Summary of Contributions

We advanced security at all levels in the computer systems stack. The table on the right provides a summary of these advances starting from hardware. Works recognized with academic prizes (best paper award, best poster award, invited articles etc.) are marked with *. While we studied the traditional trade-offs involved in implementing our security techniques in software and hardware for both legacy and new systems, we also conducted vulnerability studies to inform our research.

In addition to academic honors, our project resulted in three technology transition efforts so far: two for the Symbiote technology, and one for the technology that detects hardware backdoors. Further our vulnerability research has resulted in security patches for millions of devices in the field today.

Approved for Public Release; Distribution Unlimited.

2.0 INTRODUCTION

Defensive strategy is pervasive at all levels in the animal and plant kingdom where existence is constantly threatened due to predators and environmental vagaries. At the molecular level, our genetic code is suspected to contain high-level of redundancy, at the cellular level lymphocytes offer innate protection against viruses and microbes, at the organ level, redundancy (e.g., two kidneys) and regeneration (e.g., skin cuts, lizards dropping tails under attacks) allow continuous function and recovery under attack, and organisms have amazing ability to learn from past attacks (e.g., vaccination.) In many cases, multiple organisms co-operate (e.g., microbiomes) from symbiotic relationship to provide immunity *over and above innate and adaptive immunity*. In the biological world, the attackers have also evolved many sophisticated techniques to thwart existing defenses. The most notorious of the attackers attack the immune system itself (e.g., HIV) and is difficult to destroy because it constantly changes its tertiary structure (polymorphism), which guarantees the virus a safe harbor in the host. To provide these amazing security features organisms spend nearly 30% of their energy in defense. Given the success of flora and fauna, the defensive strategies used in biological systems are certainly worth emulating.

Computer systems today, however, are minimally defensive, if at all. Some programmers who use assertions as a defensive mechanism remove them during deployment for performance reasons. Each layer of software blindly trusts the layer beneath it for security guarantees. Anomaly detection engines are usually turned off because of the high rate of false positives. Anti-virus software itself can be hijacked and turned off. Further both hardware and software remains fairly static during their lifetime making systems a sitting duck. And despite many advances in silicon technology, there is very little defensive support available at the hardware level.

We addressed these problems in the proposal. Security is a full-system property, i.e, the software, the hardware, the system configuration, and its use should all be secure for a system to be secure. Recognizing this, in the SPARCHS project, we worked across all layers of the system stack. Further, current security research is largely top-down, where the most exposed layers --- the network/ application layers --- are first secured, and the lower layers are secured as and when threats appear. Security, thus, has become an arms race to bottom. For every software mitigation strategy today, vulnerabilities in the software layer below it can be used to attack and weaken the mitigation strategy. There are many examples of such attacks in the literature including those that attack anti-virus, libraries, operating systems, hypervisors, and BIOS routines. We wanted to avoid this problem in the SPARCHS project and thus we came devised a “hardware-up” method for building secure systems. Hardware-up simply means that the hardware would be secured first against intentional attacks, then against unintentional vulnerabilities (like a well-meaning cache optimizations that leaks cryptographic keys), and finally we add hardware support for software security so it is easy to build lean security software without a large attack surface. This work compromises work on hardware design, the microarchitecture and the architecture.

Once the hardware is secure, we focused on securing the firmware, the next level to bare hardware. For securing the firmware we invented Symbiotes. Symbiotes are diversified, embedded, inline reference monitors. We also invented techniques to remove unnecessary parts of the firmware to reduce the attack surface.

The next target was the operating system. Here we focused on mechanism to reduce concurrency bugs. We also discovered and fixed Linux kernel vulnerabilities. Following this we strengthened applications (mostly legacy ones) through binary instrumentation. Specifically we created tools to implement instruction set randomization and information flow tracking on existing binaries. Concurrently we also developed compiler techniques to implement information-flow techniques and improve robustness of programs against general reliability failures.

In the rest of the document we describe techniques by the level at which they apply. Often we implemented techniques in two different levels in the system stack; for instance, we created a tool for implementing instruction-set randomization in hardware and also created a run-time binary re-writing tool for improving legacy applications. We describe the techniques in individual sections to highlight the tradeoffs involved in implementing these techniques at the respective layer.

3.0 METHODS, ASSUMPTIONS, AND PROCEDURES

A systems-level approach was taken to designing, prototyping, and evaluating the individual components. For hardware prototyping we used FPGAs and programmed in VHDL, System Verilog and System C. We prototyped software approaches on Windows and Linux machines, on x86 and ARM machines. In addition to prototyping we also conducted vulnerability studies on printers, routers, IP phones and web browsers. All vulnerabilities were responsibly disclosed to vendors or maintainers (in the case of open source software) and have resulted in patches.

In all performance measurements we used a variety of applications to test runtime overhead. Runtime overhead is highly dependent upon the application being tested, as well as the particular inputs provided to the application, and the runtime overhead of each detection technology will vary based upon that application and workload. Further, all detection technology components can be executed in concurrently and complementarily to enhance security of the system.

4.0 RESULTS AND DISCUSSION

4.1 Hardware Security

A solution to the above problem is to push the security mechanisms down to hardware, which is typically immutable. Growing on-chip transistor budgets provide the opportunity to explore this possibility. In addition to offering immutable security, there are two further advantages to implementing security mechanisms in hardware. First, hardware supported security mechanisms can be much more energy-efficient compared to software only mechanisms. Given that energy- and power-efficiency significantly influence computing today, hardware support could very well be necessary for security mechanisms to gain traction in many real world settings. Second, implementing security mechanisms can provide unmatched visibility into execution. This provides an opportunity for new security techniques.

4.1.1 Securing Hardware Design

Hardware is the root of trust in computing systems, because all software runs on it. But is the hardware we use trustworthy? How can we ensure it has not been corrupted? Can we design it so it cannot be easily corrupted? Many factors conspire to make hardware more susceptible to malicious alterations and less trustworthy than in the past, including the growing use of third-party intellectual property components in system-on-chip designs, global scope of the chip-design process, increased design complexity and integration, and design teams with relatively few designers responsible for each subcomponent. There are unconfirmed reports of compromised hardware leading to undesirable economic consequences. A nontechnical solution is to design and manufacture hardware locally in a trusted facility with trusted personnel. However, this solution is not long term or viable, as it is neither efficient nor guaranteed to be secure. As a part of this project we created the first static analysis technique for discovering hardware backdoors.

To understand how hardware can be compromised, we need to understand how hardware is designed (see Figure 1). The first few steps are similar to software design and construction, beginning with the specification of design requirements. The hardware is then designed to meet operational requirements and coded into a hardware design language (HDL) (such as Verilog) either by designers working with the company designing the chip or with code purchased as intellectual property (such as for a USB controller) from third-party vendors around the world. The next step is slightly different from software. Hardware undergoes much more rigorous validation than most software, as hardware bugs, unlike their software counterparts, are often more expensive to fix after deployment. To minimize bugs, reputable hardware companies often employ validation teams that are much larger than the design team. They work either in tandem with designers or after the fact in the case of third-party IP components. The design, with all its components, is then processed using computer-aided design (CAD) tools from commercial companies that convert the high-level code into gates and wires. When this is done, the result is a functional design that can be reviewed for security but in practice is simply sent off to a foundry for manufacture. Reviews are encumbered by the complexity of the design and pressure of time-to-market constraints. We refer to everything until compilation with CAD tools as the front end of the process and the physical design and manufacturing at the foundry as the back end of manufacturing.

Thousands of engineers may have access to hardware during its creation and are often spread across organizations and continents. Each hardware production step must be considered a possible point of attack. Designers (either insiders or third-party providers) might be malicious. Validation engineers might seek to undermine the process. CAD tools that could be applied for design synthesis prior to manufacture might be malicious. Moreover, a malicious foundry could compromise security during the back-end manufacturing process. The root of trust is the front-end design phase; without a "golden design" to send off to the foundry, hardware designers as well as end users have no basis on which to secure foundries.

As the first line of defense, we have developed the first algorithm for performing static analysis to certify designs as backdoor free and built a corresponding tool called Functional Analysis for Nearly unused Circuit Identification, or FANCI.

Key Insight: A stealthy hardware backdoor is activated when rare inputs called triggers are processed by the hardware. Since the trigger inputs are rare, the trigger-processing circuit rarely

influences the output of the hardware circuit; it switches the output of the circuit from the good sub-circuit to the malicious sub-circuit only when a trigger is received. If security evaluators can identify portions of a hardware circuit that rarely influence output, then they can narrow down the set of sub-circuits that are potentially malicious, stealthy circuits.

Boolean functional analysis helps security evaluators identify sub-circuits that rarely influence the outputs. The idea is to quantitatively measure the degree of influence one wire in a circuit has on other wires using a new metric called "control value". The control value of an input wire 1 on a wire w2, quantifies how much the truth table representing the computation of w2 is influenced by the column corresponding to w1. FANCI detects stealthy sub-circuits by finding wires with anomalous, or low control values compared to other wires in the same design.

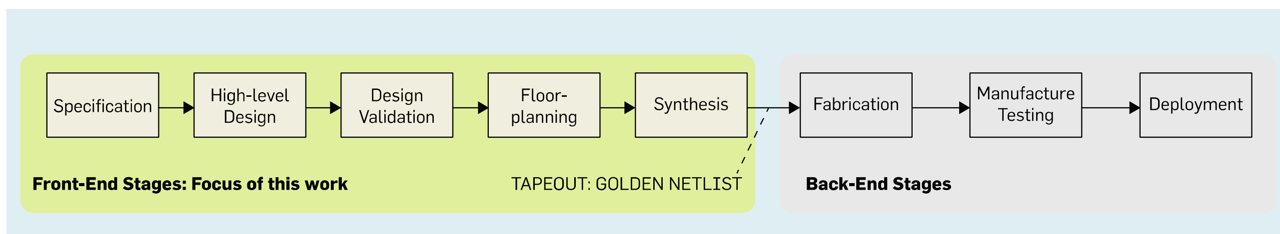

Figure 2: Stages of Hardware Design

Algorithm 1. Compute control value.

1: $count \leftarrow 0$
2: $c \leftarrow$ Column(w_1)
3: $T \leftarrow$ TruthTable(w_2)
4: **for all** Rows r in T **do**
5: $x_0 \leftarrow$ Value of w_2 for $c = 0$
6: $x_1 \leftarrow$ Value of w_2 for $c = 1$
7: **if** $x_0 \neq x_1$ **then**
8: $count$++
9: **end if**
10: **end for**
11: $result \leftarrow \frac{count}{size(T)}$

Figure 3: Algorithm for computing control value

FANCI. The algorithm to compute the control value of w1 on w2 is presented here as Algorithm 1. The control value is a fraction between zero and one, quantifying what portion of the rows in the truth table for w2 is directly influenced by w1. In step 3 of the algorithm, we do not actually construct the exponentially large truth table. Instead, we construct the corresponding Boolean function. Since the size of truth tables grows exponentially, to scale FANCI, the algorithm approximates control values using a constant-size subset of the rows in the truth table.

Algorithm 2. How FANCI flag suspicious wires in a design.

1: (Flatten design across modules if necessary)
2: (Initialize state elements to random values if necessary)
3: **for all** modules m **do**
4: **for all** gates g in m **do**
5: **for all** output wires w of g **do**
6: $T \leftarrow$ TruthTable(FanInTree(w))
7: $V \leftarrow$ Empty vector of control values
8: **for all** columns c in T **do**
9: Compute control of c
10: Add control(c) to vector V
11: **end for**
12: Compute heuristics for V
13: Denote w as suspicious or not suspicious
14: **end for**
15: **end for**
16: **end for**

Figure 4: Algorithm for flagging suspicious wires in the design

As a simple example, suppose we have a wire w2 that is dependent on an input wire w1. Let w2 have n other dependencies. From the set of possible values for those n wires (2n), we choose a constant number, say, for instance, 10,000. Then for those 10,000 cases, we toggle w1 to zero and then to one. For each of the 10,000 cases, we see if changing w1 changes the value of w2. If w2 changes m times, then the approximate control value of w1 on w2 is m ÷ 10,000. Once we have

computed all control values for a given wire (an output of some intermediate circuit), we have a vector of floating-point values we can combine to make a judgment about stealth. We find that using simple aggregating metrics (such as the arithmetic mean and median) are effective for identifying stealthy wires. Other metrics may be possible and interesting in the future. The complete algorithm used by FANCI is summarized in Algorithm 2.

In addition to running against the TrustHub Benchmark suite we performed a red team/blue team experiment, where several teams from the U.S. and Europe tried to defeat FANCI. FANCI performed well, catching all stealthy attacks and even a few non-stealthy (frequently or always-on) attacks. While FANCI is not normally expected to detect frequently-on backdoors, sometimes even frequently-on backdoors use somewhat stealthy or abnormal logic for trigger recognition.

This work received the best paper award at the CCS conference and was selected for a transition effort through Chip Scan LLC. As part of the transition effort, software was delivered to AFRL, Rome.

4.1.2 Measuring and Mitigating Micro-architectural Side Channels

The next step in our hardware-up method is to secure against unintentional micro-architectural side channels. Any computation has an impact on the environment in which it runs. This impact can be measured through physical effects such as heat or power signatures, or through how the computation consumes system resources such as memory, cache, network or disk footprints. In a side channel attack, an attacker collects these unintentional leakages to compromise the confidentiality of the computation.

A particular class of side channel attacks that rely on micro-architectural leaks has gained notoriety in the last decade. In these attacks, shared on-chip resources like caches or branch predictors have been used to compromise software implementations of cryptographic algorithms. A particularly dangerous attack demonstrated in 2009 revealed that an attacker could record keystrokes typed in a console from another co-resident virtual machine in a cloud setting by measuring cache utilization. But micro-architectural side channel dangers are not limited to cryptographic software or cloud installations: as system-on-chip designs become popular, the tight integration of components may make physical side channels more difficult to exploit. Attackers will likely turn to micro-architectural leaks to learn sensitive information.

In this area of research we a) conducted a vulnerability study to show how micro-architectural side channel attacks can be conducted remotely and even when software undergoes multiple levels of translation. The objective was to illustrate the dangers of micro-architectural side channels This vulnerability study resulted in all major browser vendors changing their browser code, b) we proposed a general-purpose mitigation for a large class of micro-architectural attacks, and c) we devised a notion of side-channel vulnerability factor to determine leakages in micro-architectural structures at run time (where they can be fixed).

4.1.2.1 Spy in the Sandbox: A vulnerability study

Even though the effectiveness of side-channel attacks is established without question, their application to practical settings is debatable, with the main limiting factor being the attack model

they assume; excluding network-based timing attacks, most side-channel attacks require an attacker in "close proximity" to the victim. Cache attacks, in particular, assume that the attacker is capable of executing binary code on the victim's machine. While this assumption holds true for IaaS environments, like Amazon's cloud platform, where multiple parties may share a common physical machine, it is less relevant in other settings. In this paper, we challenge this limiting assumption by presenting a successful cache attack that assumes a far more relaxed and practical attacker model. Specifically, in our model, the victim merely has to access a website owned by the attacker. Despite this minimal model, we show how the attacker can launch an attack in a practical time frame and extract meaningful information from the victim's machine. Keeping in tune with this computing setting, we choose not to focus on cryptographic key recovery, but rather on tracking user behavior. For our attack we assume that the victim is using a computer powered by a late-model Intel processor. In addition, we assume that the victim is accessing the web through a browser with comprehensive HTML5 support. This covers the vast majority of personal computers connected to the Internet. The victim is coerced to view a webpage containing an attacker-controlled element, like an advertisement, while the attack code itself, executes a JavaScript-based cache attack, which lets the attacker track accesses to the victim's last-level cache over time.

Using standard prime and probe algorithm with some modifications specific to Intel platforms we were able to track user behavior in private browsing mode on Safari and the Tor Browser with reasonable accuracy (80%) for the limited number of web sites. Based on the proof of concept published in the paper and a vulnerability disclosure Mozilla, Chrome, Safari and IE browsers provided a temporary countermeasure to restrict the use of high-fidelity timers in the design.

4.1.2.2 Time Warp

To close micro-architectural side-channels, broadly speaking, two strategies can be used. First, the software programmers can change the application so that there are no differences in execution times between different routines that operate on sensitive data (such as squaring and multiplication in an encryption routine). This strategy essentially slows down all operations to the slowest operation. Further, often it is difficult to identify where changes are needed. Even if these changes are made, the attacker can only be sure that only a particular known attack no longer works, and newer attacks on other parts of code may still be feasible. Second, computer architects have tried to secure micro-architectural structures themselves. For instance, if information is being leaked through a cache an architect may statically partition the cache or apply a randomized hashing scheme. This approach is reasonable, however, it requires that all leaky structures be identified in the first place. Again, while it often foils existing attacks, these protections offer little assurance against newer attacks on other shared unprotected structures. As microprocessors become more complex the number of shared structures increases (including NoC, memory controllers) and protecting them individually may prove to be difficult.

In our work we took an orthogonal approach. Rather than attempt to secure leaky structures or programs one by one -- an onerous task -- we remove attackers' ability to detect micro-architectural events. Put simply, we fuzz timing sources available to the attacker so that the attackers can no longer accurately measure fine-grained events. As a result, attackers can no longer

measure interference among contending processes and virtually all fine-grained micro-architectural attacks are foiled. This is a single simple solution that significantly raises the bar for the attacker.

The primary source of fine-grained timing information for applications is the time stamp counter (abbreviated TSC following its name in x86 architectures). The TSC is simply a free running counter that increments once every clock cycle. Simply turning off this counter would, therefore, defeat many micro-architectural attacks. Unfortunately, it would also break a large amount of existing software. Multimedia programs, games, encryption software and even the Linux kernel require RDTSC to function properly. Instead we fuzz the TSC. In doing so we follow several rules so as not to break legacy applications. In particular, we find that software expects the following of the time stamp counter:

- Monotonicity: Clocks generally only count up. Should the TSC ever appear to run backwards, negative time can be put into unsigned variables, wreaking havoc. Software generally doesn't like time going backwards.

- Entropy: Encryption software like OpenSSL often obtain entropy from randomness in the computer system through the TSC. Should we do something like mask out lower order bits of TSC, OpenSSL would be supplied with less entropy.

- Relative Accuracy: We desire to fuzz the TSC, but only a small amount -- the same order of magnitude as micro-architectural events. Applications measuring time at coarser granularities should not be severely affected. As such, speeding up, slowing down or periodically stopping time are unacceptable solutions.

- Absolute Accuracy: Applications sometimes use the TSC to get the date/time (i.e. to timestamp log messages). As such, results from TSC must always be accurate +/- our fuzz factor (which we call the wrap factor).

In short, our key idea is to divide up all time into variable sized epochs which we call the Warp factor. Within each epoch, TSC can only be read at most one time. We ensure this by adding a delay (stalling) on each read to TSC in the pipeline. We call this the real offset and it delays each TSC read into the next epoch. Additionally, we add an apparent offset to values returned by TSC. This offset changes return value to be any time within the new epoch, chosen at random. As a result of these two random offsets, attackers can read times no more often than the Warp Factor and measurements are guaranteed to be no more accurate than the Warp Factor. Further, there is no correlation between the actual time and the time measured by the user for measurements less than the Warp factor. An important aspect of our fuzzing scheme is that it makes the attacker pay a price by slowing down the attack (by stalling TSC) in contrast to prior software approaches where the user who needs security pays the price by equalizing execution times to the slowest component.

While RDTSC is by far the most common way to make fine granularity time measurements there are others. One is software timekeeping or virtual timestamp counter (VTSC) which is made possible by multi-core machines. In this scheme, a thread continuously increments a variable stored in shared memory. Disregarding scheduling and DVFS (both of which can be corrected for), this variable will increment at a constant rate, thus is a good proxy for time. Instead of reading the

hardware TSC, a thread can instead simply read this memory location. Just like TSC fuzzing we also fuzz VTSC. In short, VTSC relies on the ability of threads to communicate with very low latency. In order to break VTSC, therefore, we simply increase this latency (and make it more variable) by detecting very quick write to read communications and inserting an interrupt on the reading thread. This interrupt adds enough delay and variability to VTSC measurements that fine grained events such as cache misses can no longer be detected. This technique can be implemented on hardware available today for some performance degradation or minimal degradation with minor modifications. Using Intel's performance counters, we added an interrupt on all cache-based write to read communications on PARSEC (by monitoring the HITM performance counter). The geometric mean slowdown was measured at 4\%: this is an upper bound. We proposed a hardware modification to detect and delay only short write to read communications. This modification prevents 96% of the read-write communications from taking an interrupt in PARSEC benchmark suite. As a result, we would expect the slowdown of typical applications with this modification to be minimal.

By preventing the attacker's ability to measure information leakage, TimeWarp secures systems against a far larger set of attacks than any previous proposal. It secures against both known and as-of-yet undiscovered micro-architectural attack -- any attack which requires fine-grained measurements is thwarted. Although parts of TimeWarp can cause small performance degradations, the security implications are vast, especially in shared data centers.

Our technique does have limitations: it may be possible to average multiple runs to remove the effects of our randomization scheme. However, this increases the amount of work that an attacker has to do. Further combined with the rate-limiting effect of our fuzzing scheme (because of physical delays) it nearly makes this averaging attack infeasible. For instance, a recently published attack on cross-VM side channels took about 6 hours to extract a El-Gamal encryption key. With our protections that same attack would take nearly 190 years on a processor at 3GHz. Our solution likely puts micro-architectural side channel attack beyond the reach of most attackers. Another way this work is unique is that it gives the attacker most of the performance losses by physically slowing down the timing measurements. Prior works on security either impact the attacker or victim equally, or make the victim pay for higher penalty for security. Our mechanism thus has an inbuilt notion of punishment for the attacker.

Finally our work significantly simplifies life for a security conscious micro-architect. By creating an environment in which fine-grained micro-architectural events cannot be detected, we allow developers to design highly efficient shared micro-architectural structures and policies that would, under current standards, be considered to leak unacceptable amounts of information. Thus time warp is a simple design modification that enhances both security and also pave way for secure, higher performance architectures in domains ranging from embedded to cloud environments.

4.1.2.3 Side-Channel Vulnerability Factor

Existing processor designs often optimize for performance, power, energy or some combination thereof. However, there is growing interest in design for security. Side-channel security in particular is extremely important for a number of applications. For instance, mobile phones -- which are used for everything from authentication to accounting to entertainment systems -- hold a lot of sensitive information. Securing these processors to side channel attacks is now of

interest to both industry and academia. Being able to find leaks and quantify them is clearly an important topic.

While side-channel attacks and defenses are known before our work there was no way for a designer at design time to understand the security impact of a micro-architectural decision. Consequently designers have made micro-architectural design choices solely based on performance. We created a metric called as Side-Channel Vulnerability Factor (SVF) to guide micro-architects understand the security impact of their design decisions.

SVF is a metric and methodology for measuring the leakiness of aside-channel. It is based on our observation that there are two relevant pieces of information in a side-channel attack: the information which an attacker is trying to obtain (secret data) and the information which an attacker can actually obtain. In order to measure leakiness, we simply want to compute the correlation between these two pieces of information. Essentially, SVF represents the signal-to-noise ratio of information flowing through the side-channel; if there is a high correlation between attacker observations and the secret data, then the attacker can easily deduce the secret. If, however, there is no correlation, then the secret information is not available to the attacker.

While measuring correlation sounds easy, it is complicated by the fact that attacker's observations are not directly comparable to the secrets. For instance, in a cache side-channel the attacker measures access latencies to each cache line. These measurements effectively probe a victim's usage patterns in various cache sets. The victim's usage patterns are, of course, determined by the memory addresses used by the victim which are, it turns out, affected by things like bits in an encryption key. As a result, we wish to compute correlation between attackers' memory load latencies and the addresses which a victim is actually using. We can solve this problem with one more observation: attackers do not directly decode their observations to secrets. Rather, they look for patterns in this data. For instance, additional misses in cache set 4 may indicate that the victim encountered a 1 in the encryption key whereas misses in set 7 indicates the opposite.

These micro-architectural behavior patterns bear a strong resemblance to program phase shifts. In fact, we can consider attackers to be nothing more than phase shift detectors -- by identifying phases in victims' execution, attackers gather information about victim's inputs. Therefore, instead of directly comparing secrets data to attackers' observations, we can apply well known phase detection techniques to the problem. SVF directly measures the correlation between phases in victims' execution and phases in observations that attackers can obtain through aside-channel. Side-channel Vulnerability Factor is a methodology for measuring information leakage in a system. That is, given a system we must define SVF. SVF measurements require three components to be defined clearly:

The Victim is the application which is being attacked. SVF requires that the sensitive information in the victim be identified. For instance, memory addresses may be sensitive. Bits from encryption keys could also be used.

The Attacker is a model of an attack type. It implements a data collection method which an attacker can feasibly implement. For instance, to measure the SVF of cache systems, a generic cache scan attacker could be used.

The System itself is whatever hardware is being shared by the victim and the attacker. For instance, it could be a processor.

To compute SVF, we simply run the victim and attacker on the target system. During execution, we record the victim's sensitive data and attacker's observations as time-series data. To measure SVF, we wish to compute the correlation between these two time-series. If the attacker's observations have been influenced by the victim's sensitive data, we will observe a non-zero correlation. The greater that influence and less noise in the observations, the higher the correlation. If, however, an attacker's observations contain no trace of the secret data, the correlation will be zero or very close to zero. We cannot, however, simply compute a simple correlation (like Pearson correlation coefficient) because the two time-series have different data types -- they are likely to be vectors representing things like memory addresses or cache line misses. Instead, we analyze each time-series independently, searching for patterns. In particular, we run the same type of pattern detection used in the popular SimPoint -- comparing each data point to every other data point. This reveals repetitious behavior -- phases, in other words. This pattern analysis results in a matrix for each time-series. We can then compute a simple Pearson correlation coefficient between the two matrices.

Conclusion: SVF is a novel method for identifying application interference and leakage of sensitive data which helps us discover new vulnerabilities. From our case study, we find several general rules. Although they are derived from and specific to our simulation infrastructure, they strongly motivate the use of a quantitative metric for side-channel security: 1) Any shared structure can leak information. Even structures intended to protect against side channel leakage can increase leakage. 2) No single cache design choice makes a cache absolutely secure or completely vulnerable. Although some choices have larger effects than others, several security-conscious design choices are required to create a secure shared system. 3) The leakiness of caches is not a linear combination of design choices. Some features leak information in some configurations but protect against it in others. Others only offer effective protection in certain situations. Predicting this leakiness is, therefore, extremely difficult and probably requires simulation and quantitative comparison like did in this study.

4.1.3 Architectural Support

4.1.3.1 A Hardware Anti-Virus

Most defenses to date focus on detecting malware using features in the upper layers of the system stack. Recent works have shown promise in detecting malware programs based on their runtime micro-architectural execution patterns. Compared to their higher-level counterparts like OS and application observables, these micro-architectural features are more efficient to audit and harder for adversaries to control directly in evasion attacks.

In this work, we advance the use of these hardware supported lower-level features to detecting malware exploits in anomaly-based and signature-based detectors to detect a wider range of malware, even zero days. To perform the data collection and monitoring at low overheads, we leverage widely available hardware performance counters (HPC) in modern commodity processors (both x86 and ARM). This further enables us to protect user programs transparently without requiring modifications.

Research has shown that programs, be it malicious or benign ones, exhibit regular, reproducible behavior at the micro-architectural level. While these execution signatures vary to some degree in identical or very similar programs, they can differ radically across different types of programs. Based on this observation, we empirically demonstrate the feasibility of detecting Android malware programs given variant samples from known malware families.

We collect HPC measurements from both Android malware and typical Android good-ware, and train models that describe what constitutes malicious and benign behavior using a series of supervised machine learning techniques (such as Decision Trees, Artificial Neural Networks). Using measurements collected from the execution of another different testing set of malware and good-ware for the evaluation of the trained models, we observe we can correctly detect up to 90% of malicious malware packages with a less than 5% false positive rate.

The key intuition for the anomaly-based detection of malware exploits stems from the observation that the malware, during exploitation, alters the original program flow to execute peculiar non-native code in the context of the victim program. Such unusual code execution tends to cause perturbations to the dynamic execution characteristics of the program. If these perturbations are observable, they can form the basis of detecting malware exploits.

As we show empirically, the micro-architectural characteristics of benign programs are noisy, and the deviations exhibited by malware exploits are indeed minute. We demonstrate that with careful selection and extraction of the features combined with unsupervised machine learning, we can build baseline models of benign program execution and use these profiles to detect deviations that occur as a result of malware exploitation. In a series of experiments, we systematically evaluate the detection efficacy of the models over a range of operational factors, events selected for modeling and sampling granularity. For IE exploits, we can identify 100% of the exploitation epochs with 1.1% false positives. Since exploitation typically occurs across nearly 20 epochs, even with a slightly lower true positive rate, we can detect exploits with high probability. These are achieved at a sampling overhead of 1.5% slowdown using sampling rate of 512K instructions epochs.

We also examine the limits and challenges in implementing this approach in face of a sophisticated adversary attempting to evade anomaly-based detection. We model mimicry attacks that craft malware to exhibit event characteristics that resemble normal code execution to evade our anomaly detection models. With generously optimistic assumptions about attacker and system capabilities, we observe that the models can be susceptible to the mimicry attack with a worst-case deterioration of up to 6.5% in detection performance. The proposed detector is complementary to previously proposed signature-based detectors and can be used together as part of an ensemble of detectors to improve security.

4.1.3.2 Instruction Set Randomization

Instruction Set Randomization (ISR) was proposed in the last decade as a countermeasure against code injection attacks. ISR involves "randomizing" the ISA, thus giving the appearance of a unique instruction set for every target program. This prevents an unauthorized party (attacker) from

using the same exploit on all machines -- any code has to be in the ISA of the host program to be effective.

Current mechanisms against code-injection, such NX and its variants, are fairly effective. The fundamental drawback preventing ISR's wider acceptance stems from the fact that code-injection has ceased to be a major tool for system subversion. Increasingly modern attacks are employing code-reuse attacks to gain a foothold in the system, from which to launch other attack mechanisms, code-injection being one of them. Once such a foothold has been established, NX-bit can be bypassed and in the same manner, so can ISR. ISR, by itself, is completely ineffective against code-reuse attacks (CRA), since CRAs use legitimate code already present in a program. As such, it is powerless against one of the major modern attack vectors. Additionally, prior ISR implementations had other significant problems that challenged its practicality. Unfavorable performance-security tradeoff: Since instructions are decrypted at runtime, the decryption process falls squarely in the critical path of instruction fetch and execution (exactly where depends on the implementation). As a result, the associated latency is added to it. To offset this latency, weak encryption schemes were used to avoid impractically high performance overheads. Consequently, many attacks have been published against these schemes.

Security scheme itself prone to attack: Since most previous solutions were software based, they exposed a large TCB. Additionally they could also be turned off easily since the enabling framework did not run with extra privileges.

Required un-scalable software design: They provide limited or no support for shared libraries and page-sharing due to their restrictive trust models, although disallowing page-sharing of libraries among applications has been shown to incur impractically large memory overheads.

To counter the above problems, we have implemented Polyglot, a hardware-assisted instruction randomizing scheme with a small, untrusted software component that allows us to address all the aforementioned concerns effectively and more. By combining ISR with basic fine-grained randomization, Polyglot not only significantly improves upon the traditional security properties of ISR, it also counters state-of-the-art code-reuse attacks, which are an entirely novel and more relevant target for this technique. It utilizes strong encryption techniques (AES and ECC, in our implementation) while successfully overcoming challenges of impractical performance. Unlike most prior work, we encrypt at the page instead of application granularity. This not only enables richer diversification, but also allows us to trivially support page sharing and seamlessly scale its application to the OS, and conceptually to the hypervisor as well. With Polyglot, we also extend ISR to operate from system boot, so hardened code is available from the very first instruction the system executes. Our hardware prototype implements Polyglot on a modified Leon3-based SPARC32 processor, that runs Linux 3.8.

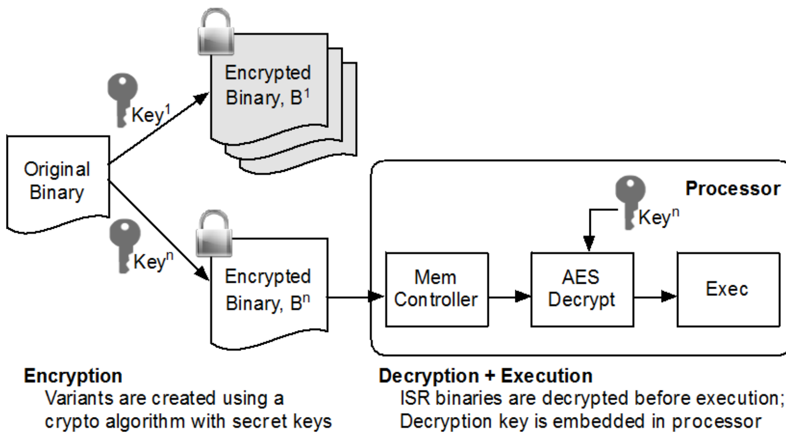

Figure 5: ISR Encryption Process

All ISR-enabled processors have an asymmetric key-pair associated, with the private key never leaving the chip confines. To prepare a binary for an ISR system, its code sections are symmetrically encrypted (AES in our case). The target processors public key is then used to encrypt the symmetric keys, which are then embedded inside the binary itself in a separate section. Upon execution the OS and loader extract the key-mappings from the binary and set up the page appropriately in the manner expected by the hardware. Note that since the keys are themselves encrypted, anyone with access to the binaries (including the OS and loader) cannot read extract the plaintext keys, and therefore, the plaintext code. On encountering a fault to a code, the MMU walks the page table and finds the corresponding ISR PTE consisting of the translation and the asymmetrically-encrypted key. At this point, it sequentially fetches the actual translation as well as the encrypted key. The key is decrypted by an ECC module and deposited into the ITLB with the original translation. Here onwards, all code originating from this page is decrypted with this key, before being stored in the I-cache. As long as this instruction is not evicted, all execution uses the decrypted instruction from this point.

Although our design only has an overhead associated with ITLB misses on regular systems, our prototype, due to its minimal nature, additionally incurs some overhead on each I-cache miss. In other words, for regular systems with multi-level cache the I-cache miss overhead would be zero. On an average, the SPEC benchmarks exhibit a 38.4% slowdown on the 16kB cache. Increasing the cache size not only reduces the number of I-cache misses drastically, it also reduces the number of ITLB misses as a result. This results in much shorter runtimes so that now the benchmarks are only 5.52% slower. The most marked reduction is observed in Omnetpp which goes from 3x to 12.5% slowdown. The reason is the drastic reduction of ITLB misses in this case. We also see speedups for some benchmarks. This is probably due to the fact that encrypting a binary involves moving around code sections a bit for alignment, and this somehow interfered with the instruction access pattern favorably.

4.1.3.3 Information Flow Tracking

Dynamic Information Flow Tracking (DIFT) is a valuable system primitive that finds widespread use in security, privacy, and program analysis applications. For example, DIFT has been used to ensure that private data does not leave a smart phone, detect security attacks such as SQL injection or buffer overflows or identify fault locations in programs when they fail. To support DIFT in a computing system each data item in a program is enhanced to include a tag that identifies some property of that data item. Then during program execution, as old data items are modified the properties of their tags are also modified, or as new data items are produced they get new property

tags according to some DIFT policy. The specific policy for creating and propagating tags is based on how DIFT is used: in a privacy application, for instance, data from the GPS receiver may be tagged as confidential, and this data and derivatives may be unsafe to leave the phone through any network interface. The size of the tags used in DIFT can vary widely depending on the application, and range from 1-bit taint tags for security to multi-byte object tags that specify data type or object evanescence.

The central question we address in this paper is: What SoC platform architecture will allow us to easily integrate DIFT support? The question of platform architecture for DIFT had not been addressed previously. The main focus before our work in the DIFT research area has been how to enhance processor architectures with DIFT. In our work we asked how we can design the DIFT mechanism at the platform level so that it is simple for third-party IP components, be it accelerators, controllers or even special cores, to be easily integrated in the SoC without intrusive changes. Towards this goal, we provided a set of recommendations for platform designers to implement DIFT as a general hardware service.

The capability we provide can be explained with a simple but realistic example. Let us say we want to build a SoC with DIFT support. Assume that our SoC only has a general-purpose core and a controller, say a DMA engine. Let us also say that on this system the data and tags for the data are stored in different locations in DRAM memory (for efficiency reasons). If the DMA engine is unaware of the separation of tags and data it will miss the tags associated with the data during copies and thus break information flow tracking. Clearly the DMA engine needs to be aware of the tag storage mechanism, i.e., it should know how to compute the address of the tags given the data address. Now, in our simple SoC, instead of a DMA engine, let us say we had a compression accelerator (or any other computational accelerator that modifies the input data). In addition to being aware of the tag storage, it should also be capable of propagating the tags through the data path within the accelerator. In this paper, we show how to extend the platform architecture so that any SoC component can easily find the tags stored in memory; the issue of tag propagation is orthogonal and not described in this work.

To easily integrate third-party IP components in a DIFT aware platform we propose a new architecture called WHISK. In our architecture the data and tags are stored separately in memory to keep a low area overhead and improve flexibility.

(a) Implicit Addressing of Tags and Data: We propose an architecture in which a NoC client on the SoC does not have to know anything about the tag layout or storage. Instead of sending a pair of addresses to access a data and its associated tag, which forces the clients to know the association mechanism between data and tags, in WHISK we allow clients to send only the data address and automatically receive or send the requested data along with its associate tag in the same packet. This strategy lowers the complexity of adapting DIFT to IP components since tags are automatically and transparently accessed with data. Further since the tag calculation is isolated from the clients, the system supports flexible tag layout and storage in memory, allowing DIFT to be easily customized for different applications.

(b) Atomic transmission: While the data and tags are stored separately in memory to keep a low area overhead, they are transported together from memory through the interconnect instead of being fetched separately as is done in single processor DIFT implementations. This coupled atomic

transport decreases the complexity of adapting accelerators to DIFT by avoiding subtle memory coherence and consistency problems between tags and data.

(c) Pipelined transfer: In our WHISK NoC protocol we send the data from/to memory one cycle after the tag. This has three main benefits. First it reduces the area overhead and design complexity since the data and tags can be sent on the same interconnect. Second the tags are already available at the clients when the data arrives at the client mitigating or completely avoiding serialization latencies during DIFT processing. Finally, since the tag and data use the same interconnect, the tag can be arbitrarily large: it can be as large as the data or if needed even larger by sending the tag over multiple packets. This allows flexible implementations of DIFT policies.

(d) Configurable, multi-granular caching: In DIFT applications, often large portions of nearby data items tend to have the same tag properties. This property can be used to reduce the area overhead of tags by representing common properties for many addresses using one tag instead of one tag per address. WHISK supports this multi-granular tag optimization. Further, in WHISK we allow clients to cache these tags to allow temporal reuse of tags to avoid latency overhead of tag accesses. These caches are also implicitly addressed with data.

(e) Standard wrapper for SoC clients: Finally, and perhaps most importantly, we show how all of the above features -- implicit addressing, atomic transmission, pipelined transfer and tag caching -- can be built in a way that allows these functions to be wrapped around existing clients in the SoC with minimum changes to the NoC or the SoC memory architecture. (Figure below). Our wrappers also handle OS interrupt processing. The wrappers are placed on the path between the NoC and the clients.

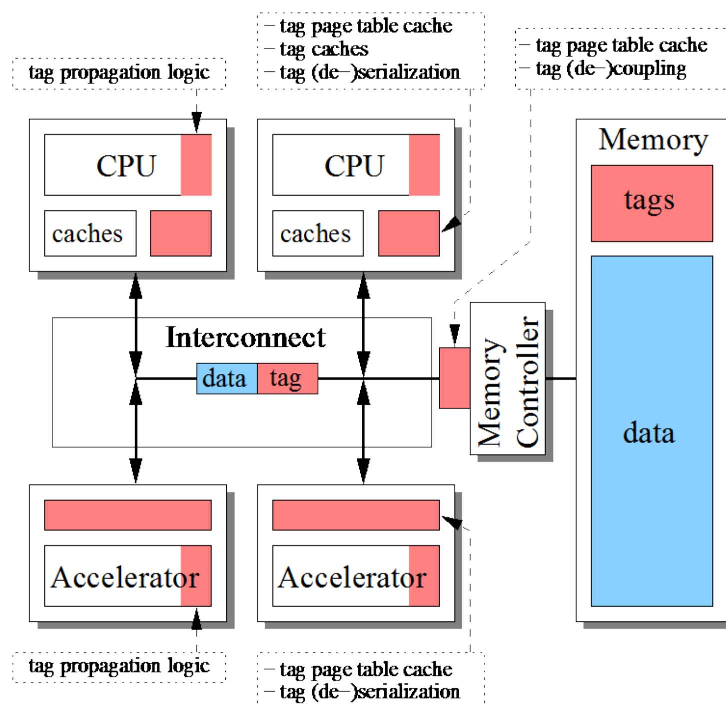

Figure 6: DIFT modifications in WHISK

To examine the practicality of WHISK we developed a cycle accurate SoC in SystemC. We were able to integrate different types of accelerators with DIFT into the system (e.g., compression, cryptography). We were also able to boot an embedded Operating System and run full applications. To test the utility of DIFT as a service we measure the impact of DIFT for different amounts of tagging by varying the fraction of the program's input data that can be tagged, and the width of the tags. This is different from prior works where overheads of DIFT were measured for specific applications of DIFT such as buffer overflows. Our experimental results with micro-benchmarks show that WHISK exhibits security-proportionality: the performance overhead is relatively proportional to the amount of tagging in the system. When running full software applications, however, the performance overhead stays almost constant, i.e., is

less impacted by the amount of tagging, because the cost of WHISK is amortized with micro-architectural optimizations, and also because of tag aggregation and caching. Finally, when active but not used, i.e., when the amount of tagging is null, the overhead of WHISK is negligible.

4.2 Firmware Security

4.2.1 Router Vulnerability Study

Cisco devices running IOS firmware constitutes a significant portion of our global communication infrastructure. Recent works demonstrate that there are vast numbers of these unsecured, vulnerable routers on the Internet. While various exploitable vulnerabilities have been reported in public, the diversity and close-source nature of embedded device hardware and firmware is touted to create a effective deterrent against practical and widespread exploitation.

In this work, we show that a new class of version-agnostic attacks is feasible by demonstrating two different reliable shellcodes that operate correctly over many Cisco hardware platforms and all known IOS versions. We develop a novel two-phase attack strategy against Cisco routers and the use of offline analysis of existing IOS images to defeat IOS firmware diversity. The key intuition behind the technique is that some IOS invariant exists in spite of the firmware diversity. The first half of the attack leverages some IOS invariant to compute a host fingerprint, and injects a stage-two shellcode to exfiltrate the host fingerprint back to the attacker. The second half of the attack then consists of a version-specific persistent rootkit with covert command and control capability.

A key contribution of this work is the conception of this new IOS rootkit that hijacks all interrupt service routines within the router. This rootkit intercepts and modifies process-switched packets just before they are scheduled for transmission. This ability allows the attacker to use the payload of innocuous packets, like ICMP, as a covert command and control channel. The same mechanism can be used to stealthily exfiltrate data out of the router, using response packets generated by the router itself as the vehicle.

Furthermore, we present the implementation and quantitative reliability measurements by testing both shellcode algorithms against a large collection of IOS images. As our experimental results show, the techniques proposed in this work can reliably inject command and control capabilities into arbitrary IOS images in a version-agnostic manner. This work underscores the importance of an effective host-based defense for routers to maintain the integrity of our global communication infrastructures.

4.2.2 Printer Vulnerability Study

In this work, we present firmware modification attacks, a general strategy that is well suited to the exploitation of embedded devices. This strategy makes arbitrary, persistent changes to victim devices' firmware by leveraging design flaws commonly found within embedded software. Firmware modification attacks can affect entire families of devices adhering to the same system design flaw, transcending operating system versions and instruction set architectures.

To demonstrate such attacks in practice, we present techniques to exploit such vulnerable functionality and the implementation of a proof-of-concept printer malware capable of network reconnaissance, data exfiltration and propagation to general-purpose computers and other embedded device types. We present a specific case study of the HP-RFU (Remote Firmware Update) LaserJet printer firmware modification vulnerability, which allows arbitrary injection of malware into the printer's firmware via standard printed documents.

We presented the results of exhaustive scans of IPv4 to track the size and distribution of all publicly accessible vulnerable LaserJet printers over time. Out of over 90,000 vulnerable units, only 1.08% of the vulnerable population has been patched since the release of firmware updates in response to the disclosure of HP-RFU. Furthermore, 24.8% of all patched printers are configured to have open telnet interfaces with no root password. In other words, we only identified 766 printers out of over 90,000 units that are simultaneously not vulnerable to the HP-RFU attack and have properly configured root passwords.

The scientific evidence, quantitative analysis and the proof of concept HP-RFU vulnerability exploitation presented in this work demonstrate the importance of introducing effective host-based defense into vulnerable embedded devices

4.2.3 Symbiotic Embedded Machines

A large number of embedded devices on the internet, such as routers and VOIP phones, are typically ripe for exploitation. Little to no defensive technology, such as AV scanners or IDS's, are available to protect these devices.

To plug this defensive gap for commodity embedded devices, we propose a host-based defense mechanism, which we call Symbiotic Embedded Machines (SEM) specifically designed to inject intrusion detection functionality into the firmware of the device. A SEM or simply the Symbiote, may be injected into deployed legacy embedded systems with no disruption to the operation of the device. We devise a Symbiote as a code structure that can be embedded in situ within the firmware of an embedded system.

The Symbiote can tightly co-exist with arbitrary host executables in a mutually defensive arrangement, sharing computational resources with its host while simultaneously protecting the host against exploitation and unauthorized modification. This has two main advantages. First, the Symbiote has full visibility into the code and execution state of its host program, and can either passively monitor or actively react to the observed events at runtime. Second, s
functions together with the host firmware, it is extremely challenging for the attacker to disable the Symbiote without rendering the device non-functional.

Furthermore, no two instantiations of the same Symbiote is the same. Each time a Symbiote is created, its code is randomized and mutated, and is stealthily embedded in a randomized fashion within an arbitrary body of firmware to protect itself from removal. This makes signature-based detection methods and attacks requiring predictable memory and code structures within the Symbiote ineffective.

Using a specific SEM implementation we call Doppelganger, we were able to automatically inject a rootkit detection payload into a Cisco 7120 router running multiple firmware images across two major IOS versions, 12.2 and 12.3. By injecting fewer than 1400 bytes of code into the IOS firmware, Doppelganger protects the router from all function hooking and interception attempts. Our white-list based rootkit detection payload does not require a priori knowledge of IOS internals, or signatures of known rootkits, and can protect the router against any code modification attempts. We demonstrate that the Symbiote injected in situ into Cisco IOS incurs negligible performance penalty and does not impact the routers functionality.

Due to the unique nature of network embedded devices, we posit that retrofitting these widely deployed vulnerable devices with defensive SEM's is the best hope of mitigating a significant emerging threat on our global communication infrastructure. SEM is a generic defensive mechanism suitable for general-purpose host protection. This research demonstrates the advantages of the Defensive Mutualistic paradigm and Symbiotes over traditional AV solutions.

4.3 Operating System Security

4.3.1 KGuard

We focused mainly on the kernel / user space boundary violations. Notably, we have identified a new, major threat for the security of the kernel and developed corresponding countermeasures. The operating system (OS) kernel has become an increasingly attractive target for attackers. This is basically due to the weak separation between user and kernel space; direct transitions from more to less privileged protection domains are permissible, even though the reverse is not. As a result, bugs like NULL pointer dereferences that would otherwise cause only system instability, become serious vulnerabilities that facilitate privilege escalation attacks. When successful, these attacks enable local users to execute arbitrary code with kernel privileges, by redirecting the control flow of the kernel to a user process. Such return-to-user (ret2usr) attacks have affected all major OSs, including Windows, Linux, and FreeBSD.

We developed a lightweight solution to the problem, which we call kGuard. kGuard is a compiler plugin that augments kernel code with control-flow assertions (CFAs), which ensure that privileged execution remains within its valid boundaries and does not cross to user space. This is achieved by identifying all indirect control transfers during compilation, and injecting compact dynamic checks to attest that the kernel remains confined. When a violation is detected, the system is halted by default, while a custom fault handler can also be specified. kGuard is able to protect against attacks that overwrite a branch target to directly transfer control to user space, while it also handles more elaborate, two-step attacks that overwrite data pointers to point to user-controlled memory, and hence hijack execution via tampered data structures.

Finally, we introduced two novel code diversification techniques to protect against attacks that employ bypass trampolines to avoid detection by kGuard. A trampoline is essentially an indirect branch instruction contained within the kernel. If an attacker manages to obtain the address of such an instruction and can also control its operand, he can use it to bypass our checks. Our techniques randomize the locations of the CFA-indirect branch pairs, both during compilation and at runtime, significantly reducing the attackers' chances of guessing their location.

4.3.2 Ret2dir: Re-thinking Kernel Isolation

As we described earlier, return-to-user (ret2usr) attacks redirect corrupted kernel pointers to data residing in user space. In response, several kernel-hardening approaches have been proposed to enforce a more strict address space separation, by pre- venting arbitrary control flow transfers and dereferences from kernel to user space. Intel and ARM also recently introduced hardware support for this purpose in the form of the SMEP, SMAP, and PXN processor features. Un- fortunately, although mechanisms like the above prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of implicit sharing still exist due to fundamental design choices that trade stronger isolation for performance.

Although the above mechanisms prevent the explicit sharing of the virtual address space among user processes and the kernel, conditions of implicit data sharing still exist. Fundamental OS components, such as physical memory mappings, I/O buffers, and the page cache, can still allow user processes to influence what data is accessible by the kernel. We studied the above problem in Linux, and exposed design decisions that trade stronger isolation for performance. Specifically, we presented a new kernel exploitation technique, called return-to-direct-mapped memory (ret2dir), which relies on inherent properties of the memory management subsystem to bypass existing ret2usr protections. This is achieved by leveraging a kernel region that directly maps part or all of a system's physical memory, enabling attackers to essentially "mirror" user-space data within the kernel address space.

The task of mounting a ret2dir attack is complicated due to the different kernel layouts and memory management characteristics of different architectures, the partial mapping of physical memory in 32-bit systems, and the unknown location of the "mirrored" user-space data within the kernel. We presented in detail different techniques for overcoming each of these challenges and constructing reliable ret2dir exploits against hardened x86, x86-64, AArch32, and AArch64 Linux targets. To mitigate the effects of such attacks, we designed and implemented an exclusive page frame ownership scheme for the Linux kernel, which prevents the implicit sharing of physical memory among user processes and the kernel. The results of our evaluation show that the proposed defense offers effective protection with minimal runtime overhead.

4.4 Application Security

4.4.1 Information Flow Tracking using Binary Rewriting

Our work on this area, involved the development of techniques that aim to a) reduce the overhead of the DFT approach (most implementations suffer from a high computational overhead), and b) detect integer overflow defects in an efficient manner.

We developed libdft, a meta-tool in the form of a shared library that implements dynamic DFT using Intel's Pin dynamic binary instrumentation framework. libdft's performance is comparable or better than previous work, incurring slowdowns that range between 1.14× and 6.03× for command-line utilities, while it can also run large server applications like Apache and MySQL with an overhead ranging between 1.25× and 4.83×. In addition, it is versatile and reusable by

providing an extensive API that can be used to implement DFT-powered tools. Finally, it runs on commodity systems. In addition, our implementation works with x86 binaries on Linux, and it can be easily extended to run on 64-bit architectures and the Windows operating system (OS). libdft introduces an efficient, 64-bit capable, shadow memory, which represented one of the most serious limitations of earlier works, as flat shadow memory structures imposed unmanageable memory space overheads on 64-bit systems, and dynamically managed structures introduce high performance penalties. More importantly, libdft supports multi-process and multithreaded applications, by trading off memory for assurance against race conditions, and it does not require modifications to programs or the underlying OS.

We then developed a novel optimization approach to dynamic DFT, based on combining static and dynamic analysis, which significantly improves its performance. Our methodology was based on separating program logic from taint tracking logic, extracting the semantics of the latter, and representing them using a Taint Flow Algebra. We have applied multiple code optimization techniques to eliminate redundant tracking logic and minimize interference with the target program, in a manner similar to an optimizing compiler. We relied upon the rich theory on basic block optimization and data flow analysis, done in the context of compilers, to argue the safety and correctness of our algorithm using a formal framework.

We evaluated the correctness and performance of our methodology by employing the aforementioned framework: libdft. Additionally, we showed that the code generated by our analysis behaves correctly when performing dynamic taint analysis (DTA). We evaluate the performance gains achieved by our various optimizations using several Linux applications, including commonly used command-line utilities (bzip, gzip, tar, scp, etc.), the SPEC CPU 2000 benchmarks, the MySQL database server, the runtimes for the PHP and JavaScript languages, and web browsers. Our results indicate performance gains as high as 2.23×, and an average of 1.72× across all tested applications.

To improve the performance of dynamic DFT, we utilized libdft again, to develop ShadowReplica. ShadowReplica accelerates DFT and other shadow memory-based analyses, by decoupling analysis from execution and utilizing spare CPU cores to run them in parallel. Decoupling analysis from execution to run it in parallel is by no means a novel concept. Previous work can be classified into three categories. The first is based on recording execution and replaying it along with the analysis on a remote host, or simply a different CPU. The second category uses speculative execution to run application code including any in-lined analysis in multiple threads running in parallel, and the third aims at offloading the analysis code alone to another execution thread. ShadowReplica belongs to the third category of systems. The main contribution behind the approach is an off-line application analysis phase that utilizes both static and dynamic analysis approaches to generate optimized code for collecting information from the application, greatly reducing the amount of data that we need to communicate. For running DFT independently from the application, such data include dynamically computed information like memory addresses used by the program, control flow decisions, and certain operating system (OS) events like system calls and signals. ShadowReplica focuses on the first two that consist the bulk of information.

DFT is run in parallel by a second shadow thread that is spawned for each application thread, and the two communicate using a shared data structure. The design of this structure is crucial to avoid the poor cache performance issues suffered by previous work. The code

implementing DFT is generated during off-line analysis as a C program, and includes a series of compiler-inspired optimizations that accelerate DFT by ignoring dependencies that have no effect or cancel out each other. Besides the tag propagation logic, this code also includes per-basic block functionality to receive all data required (e.g., dynamic addresses and branch decisions).

Our evaluations showed that compared to an already optimized in-lined DFT framework, ShadowReplica is extremely effective in accelerating both the application and DFT, but also using less CPU cycles. In essence, we do not sacrifice the spare cores to accelerate DFT, but exploit parallelization to improve the efficiency of DFT in all fronts. ShadowReplica is on average ~2.3× faster than in-lined DTA when running the SPEC2006 benchmark (~2.75× slowdown over native execution).

Finally as an application study we showed how DFT can be used to protect against integer overflow vulnerabilities. Integer overflow and underflow, signed conversion, and other types of arithmetic errors in C/C++ programs are among the most common software flaws that result in exploitable vulnerabilities. Despite significant advances in automating the detection of arithmetic errors, existing tools have not seen widespread adoption mainly due to their increased number of false positives. Developers rely on wrap-around counters, bit shifts, and other language constructs for performance optimizations and code compactness, but those same constructs, along with incorrect assumptions and conditions of undefined behavior, are often the main cause of severe vulnerabilities. Accurate differentiation between legitimate and erroneous uses of arithmetic language intricacies thus remains an open problem.

As a step towards addressing this issue, we have developed IntFlow, an accurate arithmetic error detection tool that combines static information flow tracking and dynamic program analysis. By associating sources of untrusted input with the identified arithmetic errors, IntFlow differentiates between non-critical, possibly developer-intended undefined arithmetic operations, and potentially exploitable arithmetic bugs. IntFlow examines a broad set of integer errors, covering almost all cases of C/C++ undefined behaviors, and achieves high error detection coverage. We have evaluated IntFlow using the SPEC benchmarks and a series of real-world applications, and measured its effectiveness in detecting arithmetic error vulnerabilities and reducing false positives. IntFlow successfully detected all real-world vulnerabilities for the tested applications and achieved a reduction of 89% in false positives over standalone static code instrumentation.

4.4.2 ROP mitigations

The wide adoption of protection mechanisms like address space layout randomization (ASLR), has given rise to a new exploitation technique, widely known as return-oriented programming (ROP). This technique, allows an attacker to circumvent non-executable page protections without injecting any code. Using return-oriented programming, the attacker can link together small fragments of code that already exist in the process image of the vulnerable application and they are known as gadgets. Each gadget ends with an indirect control transfer instruction, which transfers control to the next gadget according to a sequence of gadget addresses injected on the stack or some other memory area. In essence, instead of injecting binary code, the attacker injects just data, which include the addresses of the gadgets to be executed, along with any required data arguments.

We have developed a novel code randomization method that can harden third-party applications against return-oriented programming. Our approach is based on narrow-scope modifications in the code segments of executables, using an array of code transformation techniques, to which we collectively refer as in-place code randomization. These transformations are applied statically, in a conservative manner, and modify only the code that can be safely extracted from compiled binaries, without relying on symbolic debugging information. By preserving the length of instructions and basic blocks, these modifications do not break the semantics of the code, and enable the randomization of stripped binaries even without complete disassembly coverage. The goal of this randomization process is to eliminate or probabilistically modify as many of the gadgets that are available in the address space of a vulnerable process as possible. Since ROP code relies on the correct execution of all chained gadgets, altering the outcome of even a few of them will likely render the ROP code ineffective.

To evaluate our approach, we have implemented a prototype, which we call kBouncer. In addition, our evaluation using real-world ROP exploits against widely used applications, such as Adobe Reader, showed the effectiveness and practicality of our approach, as in all cases the randomized versions of the applications rendered the exploits non-functional. Note that, although quite effective, our approach is not meant to be a complete prevention solution, as it offers probabilistic protection and thus cannot deliver any protection guarantees. However, it can be applied in tandem with existing randomization techniques to increase process diversification. This is facilitated by the practically zero overhead of the applied transformations, and the ease with which they can be applied on existing third-party executables.

We further extended kBouncer, based on the detection of abnormal control transfers that take place during ROP code execution. This was achieved by using hardware features of commodity processors, which incur negligible runtime overhead and allow for completely transparent operation without requiring any modifications to the protected applications.

The new version of kBouncer was based on monitoring the executed indirect branches at critical points during the lifetime of a process, and identifying abnormal control flow transfers that are inherently exhibited during the execution of ROP code. In particular, the technique was built around Last Branch Recording (LBR), a recent feature of Intel processors. Relying mainly on hardware for instruction-level monitoring allows for minimal runtime overhead and completely transparent operation, without requiring any modifications to the protected applications.

Notably, this version of kBouncer can be selectively enabled for the protection of already installed applications. Besides typical ROP code, kBouncer can also identify the execution of "jump-oriented" code that uses gadgets ending with indirect or instructions. To minimize context-switching overhead, branch analysis is performed only before critical system operations that could cause any harm. To verify that kBouncer introduces minimal overhead, we stress-tested our implementation with workloads that trigger excessively the protected system functions. In the worst case, the average measured overhead was 1%, and it never exceeded 4%. As the protected operations occur several orders of magnitude less frequently in regular applications, the performance impact of kBouncer in practice is negligible. Finally, we evaluated the effectiveness and practical applicability of our technique using publicly available ROP exploits against widely used software, including Internet Explorer, Adobe Flash Player, and Adobe Reader. In all cases,

Approved for Public Release; Distribution Unlimited.

kBouncer blocked the exploit successfully, and notified the user through a standard error message window.

4.4.3 Destructive Code Reads in HeisenByte

In this work we focused on a newer class of dynamic code reuse attacks that builds the attack payload at runtime using memory disclosure vulnerabilities. To counter this threat of disclosure attacks on executable memory, researchers had proposed the idea of execute-only memory (XOM). While recent XOM-based techniques have proved effective on open-sourced programs, significant challenges exist in the protection of closed-source COTS binaries. Another complication in realizing the XOM concept arises from web browsers' use of dynamically generated Just-In-Time code.

To solve these two major limitations of prior solutions against memory disclosure vulnerabilities, we develop the concept of destructive code reads. Unlike XOM and XOM-inspired systems, which aim to completely prevent reads to executable memory, a task beset with many practical difficulties, we allow executable memory to be read, but make them unusable as code after being read. In essence, in our model, as soon as the code is read using a general-purpose memory dereferencing instruction, the copy of code in memory is garbled. Manipulating executable memory in this manner allows legitimate code to execute without false-positives and false-negatives, while servicing legitimate memory read operations for data embedded in the code.

We implement Heisenbyte to realize this destructive code read operation in practice on contemporary commodity systems. To efficiently detect read operations into executable memory and mediate on these operations, we leverage existing hardware-assisted nested paging feature widely available on commodity processors. Originally designed to improve performance of virtualization software, this virtualization hardware support turns out to be instrumental in allowing us to mark existing memory pages as execute-only. With a thin hypervisor shim driver, we can protect the underlying programs transparently without requiring any program modifications. Furthermore, we can support the protection of dynamically generated JIT code pages. Heisenbyte's novel use of destructive code reads sidesteps the problem of incomplete binary disassembly in binaries, and extends protection to both the static code in close-sourced COTS binaries, and dynamic JIT code. In addition to detecting attacks, Heisenbyte also offers the capability to gracefully terminate, instead of crashing, the process that is being targeted by the attack, and provide further alerting information regarding the attack to the user.

Our experiments demonstrate that Heisenbyte can tolerate some degree of imperfect static analysis in disassembled binaries, while effectively thwarting dynamic code reuse exploits in both static and JIT code, at a modest 1.8% average runtime overhead due to virtualization and 16.5% average overhead due to the destructive code reads. Amongst defenses that work on breaking determinism in systems, Heisenbyte represents a resolute and effective step towards stopping advanced exploits.

4.5 Concurrent System Security

4.5.1 A Vulnerability Study

Just as errors in sequential programs can lead to security exploits, errors in concurrent programs can lead to concurrency attacks. Questions such as whether these attacks are feasible and what characteristics were largely unknown. To answer this question we studied concurrency attacks and the security implications of real world concurrency errors. We catalogued concurrency attacks in the wild and presented their characteristics. We studied 46 different types of exploits and categorized them based on the duration of the vulnerabilities. Our study yields several interesting findings. For instance, we observed that the exploitability of a concurrency error depends on the duration of the timing window within which the error may occur. We further observed that attackers can increase this window through carefully crafted inputs. We also find that four out of five commonly used sequential defenses become unsafe when applied to concurrent programs.

4.5.2 Solutions

A key reason for concurrency bugs and their exploitability is that multithreaded programs have too many possible thread interleaving, or *schedules*. Even given only a single input, a program may run into excessive schedules, depending on factors such as hardware timing and OS scheduling. Considering all inputs, the number of schedules is even much greater. Finding a buggy schedule in this huge schedule set is like finding a needle in a haystack, which aggravates understanding, testing, and analyzing of programs. For instance, testing is ineffective because the schedules tested in the lab may not be the ones run in the field.

To reduce the number of schedules for all inputs, we have studied the relation between inputs and schedules of real-world programs, and made a surprising discovery: many programs require only a small set of schedules to efficiently process a wide range of inputs. Leveraging this discovery, we have proposed the idea of *stable multithreading (StableMT)* that reuses each schedule on a wide range of inputs. StableMT conceptually maps all inputs to a greatly reduced set of schedules, drastically shrinking the “haystack”, making the “needles” much easier to find. StableMT can greatly benefit understanding multithreaded programs and many reliability techniques, including testing, debugging, replication, and verification. For instance, testing schedules in such a much smaller schedule set becomes a lot more effective.

To realize StableMT, we have built three systems, TERN, PEREGRINE, and PARROT, with each addressing a distinct challenge. Moreover, to justify the benefits of StableMT, we have applied StableMT to address three reliability and security problems. We provide a summary of these systems below.

The first challenge of implementing StableMT is how to find highly reusable schedules for different inputs. The more reusable a schedule is, the fewer schedules are needed. However, finding highly reusable schedules is hard with existing static or dynamic techniques, because statically computed schedules are in general not guaranteed to work at runtime due to the halting problem, and dynamically computing schedules may be slow.

TERN, our first StableMT system, addresses the schedule-finding challenge by proposing a technique called *schedule memorization*: it first records a set of past, working schedules, it then reuses these schedules on future inputs when possible. Specifically, TERN maintains a cache of past schedules and the input constraints required to reuse these schedules. When an input arrives, TERN checks the input against the memorized constraints for a compatible schedule. If it finds one, it simply runs the program while enforcing this schedule. Otherwise, it runs the program to memorize a schedule and the input constraints of this schedule for future reuse. By reusing schedules, TERN avoids potential errors in unknown schedules.

Another advantage of schedule memorization is that it makes schedules explicit, providing flexibility in deciding when to memorize certain schedules. For instance, TERN allows developers to populate a schedule cache offline, to avoid the overhead of doing so online. Moreover, TERN can check for errors (e.g., races) in schedules and memorize only the correct ones, thus avoiding the buggy schedules and amortizing the cost of checking for errors.

To make TERN practical, it must handle server programs which frequently use threads for performance. These programs present two technical issues for TERN: (1) they often process client inputs (requests) as they arrive, thus suffering from input timing nondeterminism, which existing deterministic multithreaded systems do not handle and (2) they may run continuously, making their schedules effectively infinite and too specific to reuse. TERN addresses these two technical issues using a simple idea called *windowing*. Our insight is that server programs tend to return to the same quiescent states. Thus, TERN splits the continuous request stream of a server into *windows* and lets the server quiesce in between, so that TERN can memorize and reuse schedules across windows. Within a window, it admits requests only at fixed schedule points, reducing timing nondeterminism.

Evaluation on a diverse set of popular programs showed that TERN can reuse a small set of schedules to process a wide range of inputs. For instance, just 100 schedules for the Apache web server can process 90.3% of a 4-day trace (122K requests) from the Columbia CS department website.

The second challenge of implementing StableMT is how to efficiently make executions follow schedules without deviating. This challenge has existed in the area of deterministic execution and replay for decades. Previous work typically enforces two types of schedules: a total order of shared memory accesses (mem-schedule), and a total order of synchronization operations (sync-schedule). The mem-schedules are fully deterministic even with data races, but they are several times slower than traditional multithreading. The sync-schedules incur only modest overhead because most code is not synchronization and thus can still run in parallel, but these schedules may deviate if there are data races. Overall, despite much research effort, people can only choose either full determinism or efficiency, but not both.

To tackle this challenge, our second StableMT system, PEREGRINE, leverages the following insight: although data races exist in some programs, the races tend to occur only within minor portions of an execution, and the majority of the execution is still race-free.

We have implemented this insight in PEREGRINE. When a program first runs on an input, PEREGRINE records a detailed execution trace including memory accesses in case the execution runs into races. PEREGRINE then relaxes this detailed trace into a *hybrid schedule*, including (1) a

total order of synchronization operations and (2) a set of execution order constraints to deterministically resolve each occurred race. When the same input is provided again, PEREGRINE can reuse this schedule deterministically and efficiently.

Reusing a schedule only when the program input matches exactly is too limiting. Fortunately, the schedules PEREGRINE computes are often “coarse-grained” and reusable on a broad range of inputs. Indeed, TERN, our previous work, has shown that a small number of sync-schedules can often cover over 90% of the workloads for real programs such as the Apache web server. The higher the reuse rates, the more efficient and stable PEREGRINE is.

Before reusing a schedule on an input, PEREGRINE must check that the input satisfies the preconditions of the schedule, so that (1) the schedule is feasible, i.e., the execution on the input will reach all events in the same deterministic order as in the schedule, and (2) the execution will not introduce new races (New races may occur if they are input-dependent). A naive approach is to collect preconditions from all input-dependent branches in an execution trace. However, many of these branches concern thread-local computations and do not affect the program’s ability to follow the schedule. Including them in the preconditions thus unnecessarily decreases schedule-reuse rates.

Given an execution trace and a hybrid schedule, PEREGRINE computes sufficient preconditions using a new technique called *determinism-preserving slicing*. Precondition slicing takes an execution trace and a target instruction in the trace, and computes a trace slice that captures the instructions required for the execution to reach the target with equivalent operand values. Intuitively, these instructions include “branches whose outcome matters” to reach the target and “mutations that affect the outcome of those branches”. This trace slice typically has much fewer branches than the original execution trace, so that we can compute more relaxed preconditions. Evaluation on a diverse set of programs showed that PEREGRINE provides both determinism and efficiency, and can frequently reuse schedules for half of the evaluated programs.

The final system we built addresses deployment complexity: PARROT is a simple, deployable runtime that enforces a well-defined round-robin schedule for synchronization operations, vastly reducing the number of schedules. By default, it schedules synchronizations in each thread using round-robin, vastly reducing schedules and providing broad repeatability.

To mitigate the serialization problem that causes big slow-down, PARROT uses an insight based on the 80-20 rule: most threads spend most execution time in only a few core computations, and PARROT only needs to make these core computations parallel. Accordingly, PARROT provides a new abstraction called performance hints for developers to annotate core computations. These hints, which are intended to improve parallelism of core computations, are not real synchronization, and can be safely ignored without affecting correctness of a program. Specifically, PARROT provides two performance hint abstractions. First, a *soft barrier* encourages the scheduler to co-schedule a group of threads at given program points. It is for performance only, and operates as a barrier with deterministic timeouts in PARROT. Developers use it to switch to faster schedules without compromising determinism when the default schedules serialize parallel computations.

Second, a *performance critical section* informs the scheduler that a code region is a potential bottleneck, encouraging the scheduler to get through the region fast. When a thread enters a performance critical section, PARROT delegates scheduling to the nondeterministic OS scheduler

for speed. Performance critical sections may trade some determinism for performance, so they should be applied only when the schedules they add are thoroughly checked by tools or advanced developers. These simple abstractions let PARROT run fast on all programs evaluated, and may benefit other DMT or StableMT systems and classic nondeterministic schedulers.
Evaluation on a wide range of 108 popular programs (e.g., Berkeley DB and MPlayer), which is roughly 10X more programs than any previous StableMT or DMT evaluation, showed that, these hints were easy to add and made PARROT fast (12.7% mean overhead on 24-core machines). Moreover, by greatly reducing the number of possible schedules, PARROT increases the coverage of checked schedules in an advanced model checking tool by many orders of magnitude. Due to PARROT's simplicity and high practicality, we have made it open source for deployment at: https://github.com/columbia/smt-mc.

4.5.3 Applications

Applying StableMT to Improve Precision of Static Analysis To demonstrate the potential of StableMT, we have applied PEREGRINE to improve static analysis, a popular technique that analyzes a program and gets high coverage (e.g., covers all possible schedules) without executing code. One shortcoming of static analysis is that it suffers from poor precision: to get high coverage without executing code, static analysis has to over-approximate the huge schedule set and includes many schedules that will never occur. Therefore, static analysis often raises excessive false reports from the impossible schedules, which buries real bugs in noise.

Fortunately, StableMT can drastically shrink the schedule set for static analysis. We have created a static analysis framework [6] that leverages PEREGRINE to greatly reduce the number of possible schedules. The major contribution in this framework is a new approach called *schedule specialization* that combines the soundness of static analysis and the precision of dynamic analysis. Our insight is that not all of the exponentially many schedules are necessary for good performance. A small set often suffices, as illustrated by recent work on efficient deterministic multithreading. Based on this insight, our approach statically analyzes a parallel program over a small set of schedules, then dynamically enforces these schedules. By focusing on only a small set of schedules, we vastly improve the precision of static analysis; by enforcing the analyzed schedules dynamically, we guarantee soundness of the analysis results. Our approach is loosely analogous to previous approaches that combine static analysis and dynamic checking for memory safety, but ours aims at parallel programs.

Schedule specialization may be implemented in many ways for many parallel programming models such as message passing and multithreading; this paper presents one such implementation for C/C++ programs using the common Pthreads library. We represent a schedule as a total order of synchronizations such as lock operations, which can be efficiently enforced. To ensure that schedules are feasible, we collect them from real executions. To enforce schedules, we leverage PEREGRINE, our StableMT system which can enforce a small set of schedules on a wide range of inputs. For instance, it can use about a hundred schedules to cover over 90% of requests in a real HTTP trace for Apache. By reusing schedules, we not only make program behaviors repeatable across inputs, but also amortize the static analysis cost in schedule specialization.
Our framework has broad applications. For instance, we can build precise static verifiers (e.g., to verify error-freedom) because our verifiers need only verify a program w.r.t. the schedules

enforced. Stock compiler optimizations automatically become more effective on a specialized program because it has simpler control and data flow. Our framework can also benefit "read-only" analyses which do not require enforcing schedules at runtime. For instance, we can build precise error detectors that check a program against a set of common schedules to detect errors more likely to occur, while drastically reducing false positive rates. We can perform precise, automated post-mortem analysis of a failure by analyzing only the schedule recorded in a system log to trim down possible causes. We have built three highly precise analyses in this framework: an alias analyzer, a data-race detector, and a path slicer.

Evaluation on 17 programs, including 2 real-world programs and 15 popular benchmarks, shows that analyses using our framework reduced may-aliases by 61.9%, false race reports by 69%, and path slices by 48.7%; and detected 7 unknown bugs in well-checked programs.

Applying StableMT Techniques to Detect Programming Rule Violations

Real-world programs must obey many rules, such as assertions must succeed, allocated memory must be freed, and file updating and disk syncing must be done consistently. It is crucial to verify programs with these rules, because violating them can easily lead to critical failures such as program crashes, resource leaks, data losses, and security holes. Unfortunately, existing techniques can not precisely verify a program with high program path coverage. For example, although symbolic execution, a popular program analysis technique, can systematically explore program paths to find errors, this technique suffers from path explosion: it can rarely explore however a tiny portion of program paths, because a typical program is complicated and contains exponentially many paths. This poor path coverage makes rule violations extremely hard to check in real-world programs.

To address this problem, we have applied our program analysis techniques developed in PEREGRINE to build WOODPECKER, a rule-directed symbolic execution system. The insight behind WOODPECKER is: only a small portion of paths are relevant to rules, and the rest (majority) of paths are irrelevant and do not need to be verified. Leveraging this insight, WOODPECKER performs rule directed symbolic execution: instead of blindly exploring all paths, it first statically "peeks" into the paths and then symbolically executes only the paths relevant to the rule, greatly speeding up symbolic execution for both verification and bug detection.
To direct symbolic execution toward a rule, WOODPECKER faces three key algorithmic issues. First, given a rule, how can WOODPECKER determine what paths are redundant? Second, how can WOODPECKER work with different rules? It would be impractical if each rule requires a different symbolic execution algorithm. Third, how can WOODPECKER integrate with the clever search heuristics in existing symbolic execution systems? These heuristics absorb much dedicated research efforts and can steer checking toward interesting paths (e.g., those more likely to have bugs) when verifying all is not feasible. They have been shown to effectively increase statement coverage and detect errors. WOODPECKER solves these issues using two ideas: a simple but expressive checker interface and a sound, checker- and heuristic-agnostic search algorithm. A checker implementing the interface provides methods to inform WOODPECKER (1) which executed instructions are events and (2) which static instructions may be events regarding a rule. These methods abstract away the checker details, enabling WOODPECKER's search algorithm to be checker-agnostic.

Once WOODPECKER finishes exploring a path, it uses the checker-provided methods to determine which branches off the path should be pruned. Specifically, if an off-the-path branch cannot (1) affect any event executed in the path or (2) reach any new event not in the path, then it cannot lead to a different event sequence, so WOODPECKER prunes this branch without missing errors. This pruning is heuristic-agnostic because it is done only at the end of a path and does not interfere with the search heuristics otherwise. By pruning irrelevant branches, WOODPECKER can speed up symbolic execution exponentially because each pruned branch in principle halves the number of paths to explore.

Evaluation on 136 widely used programs showed that WOODPECKER verified 28.7% of program and rule combinations, whereas a top-of-the-line symbolic execution system verified only 8.5%. WOODPECKER detected seven new severe security violations.

Applying StableMT to Build Transparent State Machine Replication Service

State machine replication (SMR) leverages distributed consensus protocols such as PAXOS to keep multiple replicas of a program consistent in face of replica failures or network partitions. This fault tolerance is enticing on implementing a principled SMR system that replicates general programs, especially server programs that demand high availability. Unfortunately, SMR assumes deterministic execution, but most server programs are multithreaded and thus nondeterministic. Moreover, existing SMR systems provide narrow state machine interfaces to suit specific programs, and it can be quite strenuous and error-prone to orchestrate a general program into these interfaces.

We have built CRANE, an SMR system that transparently replicates server programs for high availability. With CRANE, a developer focuses on implementing her program's intended functionality, not replication. When she is ready to replicate her program for availability, she simply runs CRANE with her program on multiple replicas. Within each replica, CRANE interposes on the socket and the thread synchronization interfaces to keep replicas in sync. Specifically, it considers each incoming socket call (e.g., accept() a client's connection or recv() a client's data) an input request, and runs a PAXOS consensus protocol to ensure that a quorum of the replicas sees the same exact sequence of the incoming socket calls. CRANE uses a new technique we call *time bubbling* to efficiently tackle a difficult issue of nondeterministic network input timing.

CRANE schedules synchronizations using deterministic multithreading (DMT). This technique typically maintains a logical time that advances deterministically on each thread's synchronization. By serializing thread synchronizations, DMT practically makes an entire multithreaded execution deterministic. The overhead of DMT is typically moderate because most code is not synchronization and can still run in parallel. Specifically, CRANE leverages our prior DMT (and also StableMT) system PARROT, which incurs on average 12.7% overhead on a wide range of 108 popular multithreaded programs on 24-core machines.

We implemented CRANE by interposing on the POSIX socket and the Pthreads synchronization interfaces. It intercepts operations along these interfaces by hijacking dynamically linked library calls for transparency. It implements the PAXOS protocol atop the Libevent socket programming library for distributed consensus, and leverages our PARROT system for deterministic multithreading. Unlike prior SMR systems with narrow interfaces, CRANE's checkpoint and recovery must work with general programs. To this end, it leverages the CRIU tool to checkpoint and restore process states, and the LXC tool for file system states. An additional

benefit of using the LXC container is that CRANE isolates the replicated server program from the environment, avoiding nondeterministic systems resource contentions.
Evaluation on five widely used server programs (e.g., Apache, ClamAV, and MySQL) shows that CRANE is easy to use, has moderate overhead, and is robust. CRANE's source code is at: https://github.com/columbia/crane.

4.6 Compiler Optimizations for Security

4.6.1 Information Flow Tracking

Information flow tracking is useful for detecting information leakages in programs. It can also help protect against other security vulnerabilities such as buffer overflows. We implemented an LLVM compiler-based DIFT framework that makes use of compiler optimization such as dead code elimination, constant propagation, etc. to minimize performance overheads.

We applied our compiler-based information flow tracking framework to improve the precision of value profiling. Value profiling is a technique used to identify possible invariants in values computed by each instruction in programs, so that the compiler can optimize the program with respect to these invariants. Existing value profilers naively collect frequently observed values during profile execution with training inputs. However, results from naive value profiling often fail to reflect the value distribution of real execution owing to differences between training and real inputs.

By combining value profiling with information flow tracking, we can get more precise profiling results. Information flow tracking associates metadata with every dynamic value during program execution, and the metadata indicates if the value is computed from program invariants (e.g. constants or fixed inputs) or not. Therefore, we can precisely profile values that are actually valid across multiple program executions only and filter out false positives. We are also working on an automatic program specialization framework based on value profiles. Automatic program specialization based on value profiles is feasible only with high-precision profiling results, because otherwise specialization will harm the program performance due to the excessive number of incorrect predictions.

4.6.2 Region Based Memory Safety

The main issue with DIFT solutions is the high performance overhead in the absence of specialized hardware. Instead, we adopted a different approach for detecting information leakage in programs. The key insight is that information leakage is often a result of the violation of memory safety. Furthermore, attacks that violate memory safety to corrupt program state or gain control over the execution of vulnerable programs form a large class of security threats. In a type-unsafe language such as C, different vulnerabilities such as buffer overflows, format string attacks, etc. arise from the manifestation of information flows that are undefined by the C standard. For instance, pointer arithmetic more than one byte beyond allocation unit bounds is undefined by the C standard. Despite being undefined in the C standard, such behaviors are usually allowed during program execution, and often result in real-world security threats. Our technique, called region-based type

enforcement for C, relies on detecting these undefined information flows at runtime to prevent memory safety errors.

The crucial insight behind this approach is that static analysis often assumes that the program being analyzed conforms to the specifications of the C standard. Thus, we can use static analysis to build a model of information flows in a program, where undefined behavior with respect to the C standard may never occur. A runtime system can then enforce these statically defined information flows and detect violations that result from flows not defined by the static model.

Our system uses an approach similar to Write Integrity Testing (WIT), where all memory instructions in a program are classified statically into different sets, based on whether the pointers used in those instructions alias with each other. Instructions where pointers may alias are all put into the same set, whereas instructions whose pointers never alias are categorized into different sets. In other words, the memory used by a program is classified into different regions, with each region corresponding to memory that can be accessed by one of the aforementioned sets. Checks inserted at runtime enforce that instructions from a set can only access the corresponding memory region. Going back to our observation, the sets or regions in our system capture the notion of valid information flows in a program, whereas information flow between different sets would correspond to the manifestation of undefined behavior.

We have built a working prototype in our compiler, which automatically transforms a program to enforce region-based type safety. Our initial results show that the framework is successful in preventing different buffer overflow attacks with various attack targets. In addition, we have also explored extensions to our basic type enforcement framework to prevent more complex, real-world attacks such as the Mempodipper attack for local root privilege escalation.

One of the big issues with the WIT approach is that the sets detected by static analysis are unbalanced. Experiments show that on a set of 5 SPEC CPU benchmarks, 52.59% of the memory instructions are all classified into the same set. This means that any memory safety violations in these large sets are not detected by the region enforcement mechanism of WIT. WIT works around this issue by inserting guards around objects that are considered unsafe via conservative analysis; however, such an approach changes the memory layout of the program, and prevents programs with explicit pointer arithmetic from being used with this system.

4.7 Miscellaneous Works

Building on the experience from region-based memory safety, we designed a new security-aware architecture, which enforces security properties by leveraging static program information. The main features of the architecture are: (1) Using static memory dependence information to enforce integrity of memory operations in hardware, and (2) Enforcing conformity to the statically determined control flow graph at runtime.

Our current work involves optimizing performance in this security-aware architecture through two main approaches. In the first approach, we are trying to optimize single-threaded performance by enabling the architecture to perform various online optimizations such as induction variable elimination, copy propagation, etc. The second approach relies on supporting various

dynamic speculative parallelization schemes with low overhead through multi-threaded transactions (MTX).

In addition to implementing an assembler and instruction set simulation for the architecture, we have also implemented various dynamic optimizations and support for multi-threaded transactions in the gem5 simulator. The modifications to the simulator include modifications to the pipeline to enable the dynamic optimizations. In addition, to enable MTX, new coherent states were added for cache lines, and the ISA was modified to add new MTX instructions for beginning, committing and aborting transactions. Applications that benefit from these optimizations and hardware-based MTX are currently under investigation.

4.8 Transition Efforts

4.8.1 AFRL FANCI/CRADA Transition

The FANCI technique was transitioned to Chip Scan LLC for the purpose of prototyping and accelerating the FANCI technique for detecting hardware backdoors. A prototype of FANCI tool was supplied to AFRL Research Labs at Rome for further testing.

4.8.2 WHCA Symbiote Transition

The White House Communication Agency expressed interest in increasing the security of its telephony infrastructure by incorporating host defenses to ensure each critical phone device was protected from tampering, especially when the devices were being used in remote and potentially hostile locations. After several weeks of briefings and technical discussions, specific models of VoIP phones were selected by the WHCA and prototyping and demonstration of Symbiote protecting these devices was successfully demonstrated to the WHCA. The original scope of the project has been met.

4.8.3 Navy PLC Transition

The Symbiote technology transitioned to Red Balloon Security included a task to prototype and demonstrate Symbiote protected PLC's commonly used in onboard SCADA systems for the US Navy. This effort successfully demonstrated the feasibility of maintaining PLC performance characteristics while providing unique host based defense against unauthorized modifications of device firmware. The successful transition led to continuing work and subsequent project support from the US Navy in collaboration with other partners and a Navy R&D installation in Philadelphia.

4.9 System Development

As part of the project we built several hardware and software prototypes and demonstrations. The MIT LL red team selected some of the techniques for its full-system demonstration. Here we provide a list of systems developed as part of the project:

- FANCI: Techniques for detecting backdoors
- Instruction Set Randomization: Hardware FPGA prototype that boots and runs Linux. Everything from bootup to applications can be ISRized.
- Malware detectors: We demonstrated signature and anomaly based detection on ARM and X86 platforms at the DARPA PI meetings.
- Symbiotes: Symbiotes have been built for printers, routers, phones and SCADA devices.
- LibDFT: Instruction Flow Tracking tool was developed for commodity Systems
- PARROT was developed to protect against concurrency attacks.

5.0 Conclusions

The work on SPARCHS encompassed many branches of research in multiple areas and the effectiveness of each aspect of the system was demonstrated. These results have led to several papers published in top conferences. Our prototyping and transition efforts have opened up further lines of research inquiry.

6.0 REFERNCES

Adam Waksman, Matthew Suozzo, and Simha Sethumadhavan. 2013. FANCI: identification of stealthy malicious logic using boolean functional analysis. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security* (CCS '13). ACM, New York, NY, USA, 697-708. DOI=http://dx.doi.org/10.1145/2508859.2516654

Adam Waksman, Jeyavijayan Rajendran, Matthew Suozzo, and Simha Sethumadhavan. 2014. A Red Team/Blue Team Assessment of Functional Analysis Methods for Malicious Circuit Identification. In *Proceedings of the 51st Annual Design Automation Conference* (DAC '14). ACM, New York, NY, USA, , Article 175 , 4 pages. DOI=http://dx.doi.org/10.1145/2593069.2596666

Simha Sethumadhavan, Adam Waksman, Matthew Suozzo, Yipeng Huang, and Julianna Eum. 2015. Trustworthy hardware from untrusted components. *Commun. ACM* 58, 9 (August 2015), 60-71. DOI=http://dx.doi.org/10.1145/2699412

Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (CCS '15). ACM, New York, NY, USA, 1406-1418. DOI=http://dx.doi.org/10.1145/2810103.2813708

Robert Martin, John Demme, and Simha Sethumadhavan. 2012. TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In*Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA '12). IEEE Computer Society, Washington, DC, USA, 118-129.

John Demme, Robert Martin, Adam Waksman, and Simha Sethumadhavan. 2012. Side-channel vulnerability factor: a metric for measuring information leakage. In *Proceedings of the 39th Annual International Symposium on Computer Architecture* (ISCA '12). IEEE Computer Society, Washington, DC, USA, 106-117.

John Demme and Simha Sethumadhavan. 2014. Side-Channel Vulnerability Metrics: SVF vs. CSV. In *Proceedings of the 11th Annual Workshop on Duplicating, Deconstructing and Debunking* (WDDD '14).

Firefox Vulnerability Disclosure: https://bugzilla.mozilla.org/show_bug.cgi?id=1167489

Chrome Vulnerability Disclosure: https://code.google.com/p/chromium/issues/detail?id=506723

Safari/Webkit Vulnerability Disclosure: https://bugs.webkit.org/show_bug.cgi?id=146531

Tor Browser Vulnerability Disclosure: https://trac.torproject.org/projects/tor/ticket/1517

Adrian Tang, Simha Sethumadhavan, Sal Stolfo. 2014. Unsupervised Anomaly-Based Malware Detection Using Hardware Features. In *Research in Attacks, Intrusions and Defenses.* Springer International Publishing, Lecture Notes in Comptuer Science, 109-129.

John Demme, Matthew Maycock, Jared Schmitz, Adrian Tang, Adam Waksman, Simha Sethumadhavan, and Salvatore Stolfo. 2013. On the feasibility of online malware detection with performance counters. In *Proceedings of the 40th Annual International Symposium on Computer Architecture* (ISCA '13). ACM, New York, NY, USA, 559-570. DOI=http://dx.doi.org/10.1145/2485922.2485970

Joël Porquet and Simha Sethumadhavan. 2013. WHISK: an uncore architecture for dynamic information flow tracking in heterogeneous embedded SoCs. In *Proceedings of the Ninth IEEE/ACM/IFIP International Conference on Hardware/Software Codesign and System Synthesis*(CODES+ISSS '13). IEEE Press, Piscataway, NJ, USA, , Article 4 , 9 pages.

Kanad Sinha, Vasileios Kemerlis, Vasileios Pappas, Simha Sethumadhavan, Angelos D. Keromytis, 2014, Enhancing Security by Diversifying Instruction Sets, Columbia University Academic Commons, http://dx.doi.org/10.7916/D8V69GQG.

Ang Cui, Jatin Kataria, Salvatore J. Stolfo; "From Prey To Hunter: Transforming Legacy Embedded Devices Into Exploitation Sensor Grids;" The 27th Annual Computer Security Applications Conference (ACSAC); 2011/12/05.

Ang Cui, Salvatore J. Stolfo; "Defending Legacy Embedded Systems with Software Symbiotes;" The 14th International Symposium on Recent Advances in Intrusion Detection (RAID); 2011/09/20

Ang Cui, Salvatore J. Stolfo, Jatin Kataria; "Killing the Myth of Cisco IOS Diversity: Towards Reliable, Large-Scale Exploitation of Cisco IOS;" 5th USENIX Workshop on Offensive Technologies (WOOT); 2011/08/08

When Firmware Modifications Attack: A Case Study of Embedded Exploitation (NDSS 2013)

IP Phone Vulnerability:

https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2012-5445
https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2013-6685
https://downloads.avaya.com/css/P8/documents/100178648

Printer Vulnerability Disclosure:
https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2011-4161

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. kGuard: lightweight kernel protection against return-to-user attacks. In Proceedings of the 21st USENIX conference on Security symposium (Security'12). USENIX Association, Berkeley, CA, USA, 39-39, 2012.

Vasileios P. Kemerlis, Michalis Polychronakis, and Angelos D. Keromytis. ret2dir: rethinking kernel isolation. In Proceedings of the 23rd USENIX conference on Security Symposium (SEC'14). USENIX Association, Berkeley, CA, USA, 957-972, 2014.
ROP Prevention

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12). IEEE Computer Society, Washington, DC, USA, 601-615, 2012. DOI=10.1109/SP.2012.41

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. Transparent ROP exploit mitigation using indirect branch tracing. In Proceedings of the 22nd USENIX conference on Security (SEC'13). USENIX Association, Berkeley, CA, USA, 447-462, 2013.

Vasilis Pappas, Michalis Polychronakis and Angelos D. Keromytis. Dynamic Reconstruction of Relocation Information for Stripped Binaries. In Proceedings of the 17th International Symposium, RAID 2014. Springer International Publishing, Switzerland, 68-87, 2014. DOI: 10.1007/978-3-319-11379-1_4

Vasileios P. Kemerlis, Georgios Portokalidis, Kangkook Jee, and Angelos D. Keromytis. libdft: practical dynamic data flow tracking for commodity systems. In Proceedings of the 8th ACM SIGPLAN/SIGOPS conference on Virtual Execution Environments (VEE '12). ACM, New York, NY, USA, 121-132, 2012. DOI=http://dx.doi.org/10.1145/2151024.2151042

Kangkook Jee, Georgios Portokalidis, Vasileios P. Kemerlis, Soumyadeep Ghosh, David I. August, and Angelos D. Keromytis. General Approach for Efficiently Accelerating Software-based Dynamic Data Flow Tracking on Commodity Hardware. In Proceedings of the 19th Internet Society (ISOC) Symposium on Network and Distributed System Security (NDSS). February 2012, San Diego, CA.

Kangkook Jee, Vasileios P. Kemerlis, Angelos D. Keromytis, and Georgios Portokalidis. ShadowReplica: efficient parallelization of dynamic data flow tracking. In Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security (CCS '13). ACM, New York, NY, USA, 235-246, 2013. DOI=http://dx.doi.org/10.1145/2508859.2516704

Marios Pomonis, Theofilos Petsios, Kangkook Jee, Michalis Polychronakis, and Angelos D. Keromytis. IntFlow: improving the accuracy of arithmetic error detection using information flow tracking. In Proceedings of the 30th Annual Computer Security Applications Conference (ACSAC '14). ACM, New York, NY, USA, 416-425, 2014. DOI=http://dx.doi.org/10.1145/2664243.2664282

Adrian Tang, Simha Sethumadhavan, and Salvatore Stolfo. 2015. Heisenbyte: Thwarting Memory Disclosure Attacks using Destructive Code Reads. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (CCS '15). ACM, New York, NY, USA, 256-267. DOI=http://dx.doi.org/10.1145/2810103.2813685

Feng Liu, Soumyadeep Ghosh, Nick P. Johnson, and David I. August. CGPA: Coarse-Grained Pipelined Accelerators. The Design Automation Conference (DAC), June 2014. Link:http://liberty.princeton.edu/Publications/dac14_cgpa.pdf

Speculative Separation for Privatization and Reductions, Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2012.

Speculative Separation for Privatization and Reductions, Nick P. Johnson, Hanjun Kim, Prakash Prabhu, Ayal Zaks, and David I. August. Proceedings of the 33rd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI), June 2012.

Runtime Asynchronous Fault olerance via Speculation Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Proceedings of the 2012 International Symposium on Code Generation and Optimization (CGO), April 2012.

John Demme and Simha Sethumadhavan, Rapid Identification of Architectural Bottlenecks via Precise Event Counting, Proceedings of the 38th ACM/IEEE International Symposium on Computer Architecture.

Junfeng Yang, Ang Cui, Salvatore J. Stolfo, Simha Sethumadhavan; "Concurrency Attacks;" the Fourth USENIX Workshop on Hot Topics in Parallelism; 2012/06/07.

Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth Gibson, and Randy Bryant. “Parrot: a Practical Runtime for Deterministic, Stable, and Reliable Threads”. Proceedings of SOSP 2013.

Heming Cui, Jingyue Wu, John Gallagher, Huayang Guo, and Junfeng Yang. “Efficient Deterministic Multithreading through Schedule Relaxation”. Proceedings of SOSP 2011.

Heming Cui, Jingyue Wu, Chia-che Tsai, and Junfeng Yang. “Stable Deterministic Multithreading through Schedule Memoization”. Proceedings of OSDI 2010.

Heming Cui, Gang Hu, Jingyue Wu, and Junfeng Yang. “Verifying Systems Rules Using Rule-Directed Symbolic Execution”. Proceedings of ASPLOS 2013.

Junfeng Yang, Heming Cui, Jingyue Wu, Yang Tang, and Gang Hu. "Determinism Is Not Enough: Making Parallel Programs Reliable with Stable Multithreading". In Communications of ACM 2014.

JingyueWu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. "Sound and Precise Analysis of Parallel Programs through Schedule Specialization". Proceedings of PLDI 2012.

Heming Cui, Rui Gu, Cheng Liu, Tianyu Chen, and Junfeng Yang. "Paxos Made Transparent". Proceedings of SOSP 2015.

LIST OF SYMBOLS, ABBREVIATIONS, AND ACRONYMS

Acronym	**Nomenclature**
ACM	Association for Computing Machinery
ARM	Advanced RISC Machine
ASLR	Address Space Layout Randomization
BAA	Broad Agency Announcement
CCS	Computer and Communications Security
CPU	Central Processing Unit
CWE	Common Weakness Enumeration
DBI	Dynamic Binary Instrumentation
DBT	Dynamic Binary Translator
DEP	Data Execution Prevention
DFT	Dynamic Flow Tracking
DTA	Dynamic Taint Analysis
FCG	Function Call Graph
FTP	File Transfer Protocol
HTTP	Hyper Text Transfer Protocol
IDT	Interrupt Descriptor Table
ISR	Instruction Set Randomization
I/O	Input/Output
LOC	Lines Of Code
MD5	Message-Digest 5
MIPS	Millions of Instructions Per Second

MMU	Memory Management Unit
MP	Memory Protection
OS	Operating System
PC	Program Counter
PDF	Portable Document Format
QEMU	Quick Emulator
RISC	Reduced Instruction Set Computer
ROP	Return-Oriented Programming
SAX	Symbolic Aggregate Approximation
SEP	Symbiotic Embedded Machines
SQL	Structured Query Language
SSH	Secure Shell Host
SPEC	Standard Performance Evaluation Corporation
SVF	Side-Channel Vulnerability Factor
T&E	Test and Evaluation
VMI	Virtual Machine Introspection
VMM	Virtual Machine Monitor
VOIP	Voice Over Internet Protocol
XOR	Exclusive Or