Enhancing Security by Diversifying Instruction Sets

KANAD SINHA, Columbia University
VASILEIOS KEMERLIS, Columbia University
VASILIS PAPPAS, Columbia University
SIMHA SETHUMADHAVAN, Columbia University
ANGELOS KEROMYTIS, Columbia University

Despite the variety of choices regarding hardware and software, to date a large number of computer systems remain identical. Characteristic examples of this trend are Windows on x86 and Android on ARM. This homogeneity, sometimes referred to as “computing oligoculture", provides a fertile ground for malware in the highly networked world of today.

One way to counter this problem is to diversify systems so that attackers cannot quickly and easily compromise a large number of machines. For instance, if each system has a different ISA, the attacker has to invest more time in developing exploits that run on every system manifestation. It is not that each individual attack gets harder, but the spread of malware slows down. Further, if the diversified ISA is kept secret from the attacker, the bar for exploitation is raised even higher.

In this paper, we show that system diversification can be realized by enabling diversity at the lowest hardware/software interface, the ISA, with almost zero performance overhead. We also describe how practical development and deployment problems of diversified systems can be handled easily in the context of popular software distrbution models, such as the mobile app store model. We demonstrate our proposal with an OpenSPARC FPGA prototype.

1. INTRODUCTION

Many large-scale security attacks are facilitated by the lack of diversity in computer systems. Today many computers have the same hardware and software configuration, *e.g.,* Windows on x86 and Android on ARM, and consequently suffer from the same vulnerabilities. Even worse, security protection mechanisms, as well as their workarounds, are also invariable. This homogeneity allows an attacker to prepare malicious binaries, and deploy them quickly and profitably on a large number of victim systems.

We will demonstrate the dangers of homogeneity, and the incentives it provides for malware writers, using two representative examples. According to a report published by Symantec earlier this year, the two infection modes described below are the predominant strategies for malware infection [Symantec Corporation 2012], thus aiding creation of thriving underground malware economies [Stone-Gross et al. 2013; Trend Micro Corporation 2012].

• **Social Engineering-based Code Download** In such an attack, an unsuspecting (often lay) user is prompted to click on a URL link under a false pretense, such as the need of a codec for viewing online videos [Lavasoft Corporation 2006]. However, when the user clicks on the link, the downloaded binary does more than just displaying the web page correctly. The attacker uses it to gain a foothold on the victim’s system and use it for nefarious purposes, which range from stealing the user’s credentials, credit card numbers, and personal information, to selling the computer resources to a bot herder [Binsalleeh et al. 2010].

• **Drive-by-Download Code Injection** In a drive-by-download attack [Provos et al. 2007], the victim usually receives an attachment via email or a URL link pointing a malicious web page. When the victim opens the attachment or reaches the landing page, the attacker exploits a vulnerability in the viewing application and transfers control to a payload. Unlike the previous case, the payload here is usually encoded inside the input data stream. The process of transfer-

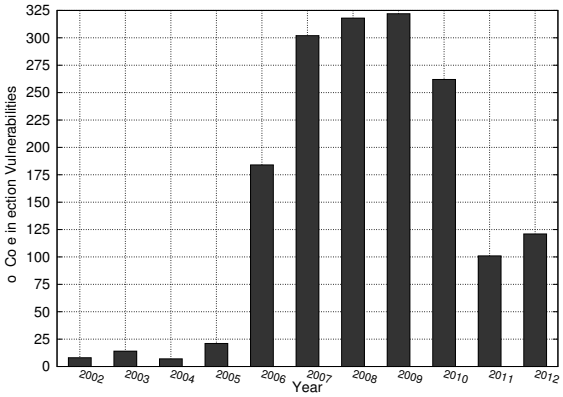

Fig. 1. Code-injection vulnerabilities during the last decade (collected from the National Vulnerability Database of NIST).

ring control to a malicious payload (typically code encoded as data) is known as code-injection [MITRE Corporation 2012].

These attacks are made possible not only due to the poor security awareness on part of the users, but also because of the large number of systems sharing the same kind of hardware/software stack, which in turn allows attackers to monetize such "write once, run anywhere" exploits.

Just like in biological systems, where diversity within an ecosystem prevents large populations from going extinct due to a single pathogen, the diversification of systems can make them more robust by making them invulnerable to generic attack payloads and infection vectors. To address the lack of diversity and concomitant exploits, computer security researchers have proposed the diversification of systems as a security strategy [Forrest et al. 1997; National Science and Technology Council 2011]. Diversification forces the attacker to develop custom exploits for each diverse entity, potentially to a point where attack creation becomes an unattractive process from a time and monetary standpoint. For example, computer systems can be diversified by having different architectural register names, instruction formats, caller/callee save conventions *etc.*

The diversification approach is in direct contrast to the prevalent security mindset that follows a "patch and pray" paradigm. That is, protections are created to handle the symptoms instead of dealing with the root cause, which is that *attackers find it profitable to deploy exploits en masse and are often successful in doing so*. An example of such a defense was the introduction of no-execute (NX) permissions on data pages to prevent code injection. Furthermore, code signing was introduced an attempt at stopping illicit code download. These protection mechanisms, however, have not been completely effective (Figure 1).

We propose *native hardware support* for diversifying systems by enabling instruction-set randomization (ISR), which aims to provide a unique random ISA for *every* deployed system (see Figure 2). For instance, the opcode `0xa` may denote the `XOR` operation on one machine, but may be invalid on another. Software implementations are too slow (70% to 400% slowdowns) and more importantly, are insecure because they use much weaker encryption schemes and can be turned off.

When implementing diversification we want to keep two things in mind however. First, we want to minimize the hardware design effort – for instance, we do not want to create a new microarchitecture for each unique ISA. That would be too onerous to be practical. Second, we want to allow the construction of legitimate software easily,

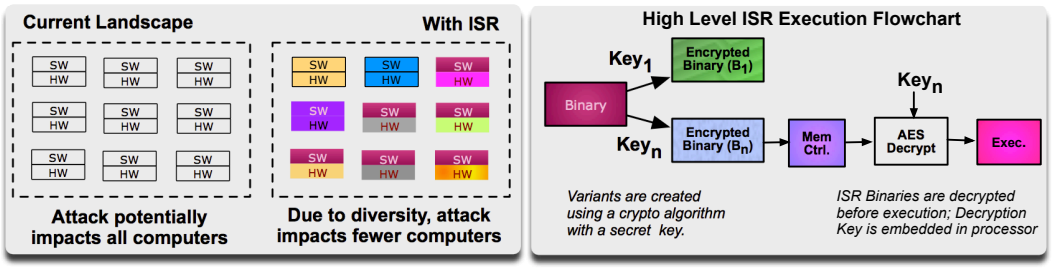

Fig. 2. High level overview of Instruction Set Randomization.

while selectively increasing the difficulty of malware creation. Our secret ISA construction and the associated software model achieves both these goals. Basically by encrypting the code with "secret" keys, we are able to emulate the benefits of having a different ISA for each computer. We leverage a strong cryptographic scheme, such as AES [of Standards and Technology 2001], to encrypt program binaries and decrypt the binary just before execution. This method for generating diversity is far simpler than actually customizing the decoder on each chip to implement random mappings or changing the microarchitecture of every instance.

In this paper, we discuss the system-level, architectural and microarchitectural design choices for enabling ISR. We outline various granularities of diversification and explore the hardware and software aspects of implementing two extreme choices in that spectrum that offer different complexity-vs-security tradeoffs. For both cases, we are able to reduce the performance overheads of ISR to zero with strategic microarchitectural optimizations. In addition, we also outline possible deployment and key management techniques, *i.e.,* how are keys embedded in hardware, how are they accessed, *etc.*, within the framework of established distribution models. This is an important aspect of any such design and is crucial because we do not want to make the production and deployment of all software difficult, just that of malware. Finally, we describe a prototype on the OpenSPARC FPGA processor, and in that context we explain how hardware-software needs to change for accommodating diversification.

To summarize, our contributions in this paper are:

(1) we perform a design space exploration for instruction set randomization and describe attendant design choices at the architectural and microarchitectural level, and show how to provide a cryptographically strong hardware support for ISR,

(2) describe an OpenSPARC FPGA implementation,

(3) outline distribution models for ISR-enabled systems.

The rest of the paper is organized as follows. In Section 2, we outline our security trust model. Section 3 discusses the general design considerations for ISR, while Section 4 explores the hardware design options with a description of our OpenSPARC prototype presented in Section 5. Section 6 presents some evaluation results, Section 7 presents our security analysis, and Section 8 outlines possible distribution models for two extreme schemes in our design space. Related work is presented in Section 9 and finally conclude in Section 10.

2. THREAT MODEL AND TRUST ASSUMPTIONS

Goal and Threats Addressed Attacks typically follow the *path of least resistance*. Therefore, by breaking systems' homogeneity and providing diverse computing platforms, we force attackers to develop custom exploits thereby drastically reducing their

return-on-investment. Currently, we only consider binary code-injection attack vectors achieved via drive-by-download and social engineering attacks, similar to those described in Section 1. In such scenarios, the attacker's goal is to lure the user into running a malicious executable or inject code on a local application (after exploiting a software vulnerability).

System Assumptions ISR is by no means a panacea against all security problems. In addition to the diversification of instruction sets that we propose in this paper, we assume the use of contemporary defense mechanisms, *e.g.,* address space layout randomization (ASLR) and software-only diversification solutions for mitigating Return-oriented programming (ROP) attacks. This assumption is reasonable since ASLR and ROP defenses are viable and immediately deployable [PaX Team 2010; Pappas et al. 2012; Kayaalp et al. 2012]. We also assume that support for hardware-aided protection exists in processors in the form of NX bits, virtualization (AMD-V, VT-x), trusted execution (Intel TXT, ARM TrustZone), *etc.* Our solution is *agnostic* and *orthogonal* to such technologies that behave homogeneously in all deployments. However, note that diversification can make them more effective, by raising the bar for an attacker that studies their deployment and tries to bypass them. Instruction set diversification is a missing piece that will nicely complement such mechanisms.

NX Technology We assume that NX protection (Intel's eXecute Disable bit, AMD's Enhanced Virus Protection, ARM's eXecute Never bit) is not always effective against binary code-injection attacks. As shown in Figure 1, code-injection has increased despite the widespread adoption of NX hardware. In Section 7 we elaborate on the reasons behind this trend.

Trusted Principal We assume a trustworthy software principal, part of our trusted computing base (TCB), to supply and manage keys in hardware. This is simply a module in the hypervisor or OS, and can supply keys to authorized processes when needed and manage the keys during execution if necessary. Existing techniques for privileged isolated execution can be used to create such a process [Azab et al. 2011].

Crypto Algorithm A strong cryptographic algorithm that is not susceptible to known-plaintext attack, such as AES, is assumed to be used for diversifying the ISA. This ensures that if an attacker has the plaintext and the encrypted binary, he will not be able to recover the encryption key.

Key Storage The security of our scheme depends on keeping the encryption keys secret. Further, each system shall use a different secret key. We also assume that remote key transfers are done via secure channels (IPSec).

3. DESIGN CONSIDERATIONS

The working principle behind ISR support is quite simple: programs on a computer should be encrypted with *secret* keys in some fashion and decrypted before execution. We want to use cryptographically strong encryption to prevent the attacker from guessing the ISA. Basically, we emulate randomization by encrypting code with different keys. There are several possible mechanisms and policies to implement this concept that offer a spectrum of diversity vs. complexity tradeoffs. In this section, we discuss some of these design options and the associated tradeoffs.

3.1. Granularity

Randomization can be obtained by applying encryption for different message lengths[1], which correspond to different granularities of code encrypted with a single key in a system.

[1]We follow the standard convention and consider that a message consists of one or more blocks. Despite similar terminologies, message blocks are not to be confused with cache blocks though.

① **One key for the entire system.** The simplest design choice is to encrypt all software (and firmware) on the system with the same key, which should also be available for decrypting instructions. Without the key the attacker's binary code (the malicious payload) will be decoded as a garbled sequence of instructions, and will not behave as the attacker intended. The obvious problem of this approach is the excessive amount of trust put on the system-wide secret key. If the key is leaked, then the complete system is useless from a security point of view.

② **One key for each privilege level.** Instead of using one key for the entire hardware/software stack, we can envision a system where there is one key per privilege level, and is automatically swapped by hardware on privilege level changes. A leaked key again means that all software in that level is compromised and has to be encrypted with a new key, which in turn has to be installed in the hardware. We would also need to ensure that resetting hardware keys is beyond all privileges (else a stolen key at the appropriate privilege level can be used for unauthorized re-keying).

③ **One key per process.** This option provides a strong degree of protection against key leaks – a leaked key for one application does not put any other application in danger. However, such an approach complicates the design of software in several ways. First, some trusted entity has to store and manage keys for potentially a very large number of applications. Further, this software component needs to operate at the operating system (OS) level, since the process abstraction is not readily available below the OS, and as such, it will be inherently vulnerable to OS kernel vulnerabilities and attacks [Spengler 2007]. Another problem with this scheme is that it breaks code-sharing among applications; shared code regions, like libraries, will be encrypted under different keys. One simple workaround would be to statically compile all binaries.

④ **One key per executable package.** Under this scheme each package containing executable code (including shared libraries) is encrypted with different keys. Consequently, a running process might have multiple static and dynamically loaded code modules (corresponding to the executable and the shared libraries) in its address space, each encrypted with a different key. This design has the same order of key complexity as the previous proposal, except now sharing common memory is possible at the granularity of executable modules, allowing the use of shared libraries as usual.

⑤ **One key per page.** Here we allow each memory page, even within the same executable or library, to have different encryption keys. This is similar to and compatible with ④. However, it gives us more diversity than the per-executable keys, since by knowing one key or figuring out a valid opcode, a malicious user can merely do code-injection within the respective memory page.

⑥ **Smaller granularities.** One could also imagine smaller encryption granularities, like functions, basic blocks, cache block width, instruction-level, *etc.* This could provide enormous diversity, but the overhead for supporting such schemes is likely to be too high to be practical.

In this paper, we choose to explore two of the aforementioned design choices—one at each extreme of the design spectrum: system ISR ①, and page ISR ⑤. We discuss the hardware design considerations for these choices in Section 4, and highlight possible models for software distribution and deployment in Section 8.

There are some commonalities, however, between the two approaches. First, we will be using a strong encryption algorithm, such as AES with 128-bit keys. Second, we want ISR to be enabled from the very first instruction executed in the processor to protect against even malicious BIOS attacks [Sacco and Ortega 2009; Wojtczuk and Tereshkin 2009], support distribution of authorized compiled binaries (and patches), and allow performance-, productivity- enhancing features such as shared library support.

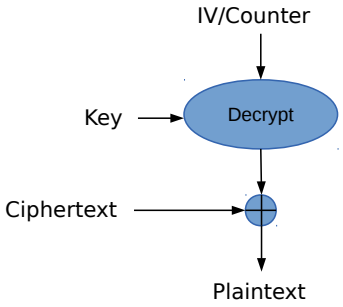

Fig. 3. Counter-mode block decryption

3.2. Encryption Mode

We choose symmetric ciphers over asymmetric ones for encryption, since the latter are too expensive. For the rest of this paper, we will assume use of AES, but practically any other secure version can be used instead.

Symmetric block ciphers can be used in a number of modes [Dworkin 2001]. An important requirement for our design is that decryption of any random block in a message should not depend on any other block. This property of random access is necessary since we want decryption of blocks independent of other blocks. If this were not so, all required blocks would have to be made available to the decryption accelerator at once by fetching them from cache or memory, which will not only increase decryption latency but also pollute the cache. This rules out CBC, CFB and OFB modes. Additionally, for stronger security we decide against ECB mode since they are prone to dictionary and block-reuse attacks. This leaves us with counter-mode. Indeed this mode has been used previously to aid secure memory encryption [Suh et al. 2003a; Shi et al. 2005; Yan et al. 2006].

In addition to the key, counter mode requires an additional argument called the initialization vector (IV). This is used as a counter seed for encryption/decryption of a message block. The same IV needs to be used for encryption as well as decryption, and need not be kept secret. An additional requirement (as recommended in [Dworkin 2001]) is that for a given key, each message block must use a unique counter value; else dictionary attacks can be mounted across ciphertext messages that use the same key and counter. Operation of counter-mode block decryption is illustrated in Figure 3. The combination step of the (en/)decrypted key with the message block is usually just a XOR operation.

4. HARDWARE SUPPORT

At the hardware level, an ISR implementation—single key or page mode—needs to address three fairly straightforward objectives:

(1) For correctness, encrypted instructions should be decrypted before execution.
(2) To mitigate penalties, instruction decryption should be as far away from the energy and performance critical paths in the processor,
(3) For security, the instructions should be decrypted only where the adversary does not have the ability to re-write the instruction stream and the key(s) used for encryption should be available only to authorized users.

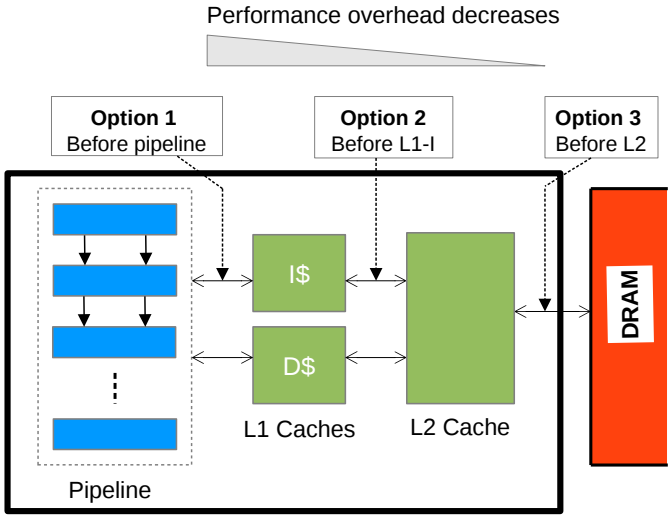

Fig. 4. Microarchitectural choices for implementing ISR.

Although system ISR requires less key storage space than page ISR, both modes share similar design tradeoffs. First we discuss the common design choices, and then the design specific modifications.

4.1. Placement of Decryption Unit

Our design requires encrypting sets of instructions corresponding to a cache block as a single unit[2]. We propose performing the decryption on every block somewhere along the CPU-memory pathway (see Figure 4).

Since the encryption block size is smaller than the standard cache block (typically 512-bits), we propose dividing the cache block into chunks and using the same key (and IV if applicable) to decipher them in parallel. For instance, for 128-bit key width the cache block can be partitioned into 4 chunks and decrypted simultaneously. Alternatively, if we use 256-bit wide key, the block has to be deciphered in halves. The lower order bits of the IV could be used to distinguish between these chunks.

Before pipeline decode stage A naive solution is to place the decryption unit inside the pipeline just before the fetch and decode stage. This design will be prohibitively energy inefficient and slow because a strong cryptographic algorithm, such as AES, will require between 20-120 cycles for decryption. While pipelining the crypto unit can mitigate some of the throughput concerns, the power and energy overheads may be significant as each instruction has to repeatedly go through the decryption unit.

Between L1-I and L2 We observe that decryption can be performed anywhere on chip as long as there is a clear distinction between instructions and data in the stream. One of the first structures in the microarchitecture where this becomes apparent is when instructions are filled from the L2 cache into the L1-I cache. If we include the decryption unit in the instruction miss fill path, we can amortize the cost of decryption due to locality available in the L1 cache. Since most cache accesses hit in the L1-I

[2]Note that this does not conflict with the encryption granularities proposed before. For instance, in case of page ISR, one key per page is used to perform encryptions, where each encryption unit is as wide as the cache block.

cache, ISR only adds to the L1-I miss penalty. If the processor supports simultaneous multithreading this latency can be mitigated even further.

At cache-memory interface Can we push the decryption process even further to mitigate the latencies? In case we choose to decrypt at the L2-memory boundary (assuming a two level cache system), the design becomes a bit more complicated. This is mainly because beyond the L1-I, we do not know whether a block is destined to be treated as data or instruction (or both), and we only want to act on instructions.

We solve this problem by learning the nature of a requested block at the microarchitectural level. When we receive a (miss) request from the L1-I, we know this request to be an instruction miss. We simply preserve the source of miss (instruction or data) all the way to the memory controller to identify instruction miss requests. When the miss reply comes in from DRAM we selectively apply decryption to the instruction fills. With this scheme instructions are decrypted before being filled into the L2, and remain decrypted in the lower levels (L2 and L1) where most of the accesses hit, significantly reducing the penalty of ISR.

However, our microarchitecture optimization of pushing decryption beyond the L1 creates a security vulnerability. Say a cache block was fetched from DRAM into the L2 in response to a data load request. Now let's say that this data is also requested by the L1-I. In this scenario, the block, which came in as data without going through the decrypting process, is now fed as is to the L1-I and eventually makes its way into the pipeline, thus completely bypassing ISR. An attack vector to exploit this might involve first loading the shell-code, crafted in native ISA, as data, and referencing the same locations for execution soon thereafter while they are still in L2. Conversely, consider the case when a block that resides in L2 is fetched as an L1-I fill, *i.e.,* as an instruction, but is eventually requested by the L1-D. The decrypted instruction (in a form which can be easily disassembled) is now being treated as data. In such a scenario, it is not hard to imagine a well-crafted attack reading off decrypted instructions from known locations. Given the location and the decrypted instruction, a reliable dictionary of instruction mapping could be easily constructed that can be used to construct valid pieces of ISRized code.

We prevent these cases by tracking instructions and data cache blocks in all caches beyond the L1. We do so by adding a bit to each cache block indicating whether it is instruction or data. This bit is set by tracking the first source of the miss, *i.e.,* the instruction or the data cache. Only blocks marked as instruction can be fed to the L1-I and vice versa for data. Cross-sharing between the split caches is thus completely disallowed, either directly or through L2. If L2 receives a request from the L1-I for a block marked data, that block has to be flushed to memory and fetched again, this time going through the decryption process. Similarly, when a block marked instruction is requested as data, it is flushed and fetched again; only now the decryption module will be bypassed during the L2 fill.

Overheads As shown in Figure 3, in counter mode, decryption is not performed on the ciphertext but on the counter. This allows simultaneous block fetch and decryption, assuming: (a) the appropriate counter value is known at request initiation, and (b) XOR operation on the block does not take any cycles. There is, therefore, a performance overhead only when the decryption process takes longer than the fetch from the next stage, and if so, the overhead is $(decryption\ latency) - (fetch\ latency)$ cycles. This also implies that the last option described above *viz.,*, decrypting at the L2-memory interface, will incur no overhead at all if decryption latency is less than memory fetch latency, which is generally the case.

4.2. Key Storage

We need to store 128-bit AES keys in hardware for both ISR modes. For system ISR, the key can be stored at the decryptor module itself since they are not going to be modified (at least during normal operation). For page ISR, a logical place to hold the per-page AES keys (and the corresponding IV if required) is the ITLB. On a page fault, the key corresponding to an encrypted page is installed into the ITLB SRAM (with or without OS assistance) as part of the fault handling mechanism. Subsequently, every time an instruction miss request is sent out the page keys are also read from the ITLB and sent out as part of the miss request so that the blocks can be unencrypted at the appropriate level.

4.3. Encryption

As discussed in Section 3.2, we will use symmetric encryption in counter mode. This means that, apart from the key, decryption process also requires a counter derived from some IV. This opens up more design choices.

Page ISR: In this scheme each code page of an application is encrypted with a unique key[3]. For each code block in a page, the counter is the block offset in the page. One advantage of this scheme is that, since the counter can be completely inferred from the virtual/physical address, the IVs need not be stored.

System ISR: In this mode there is one key for the entire system. Thus, secure counter mode requires a unique counter for *every* code block. One way of achieving this is by assigning every package a unique ID in range $[1, 2^{73}]$ [4]. The IV for each code page in a package is the concatenation of the package ID and the absolute address of the page in the package. The actual counter may be derived as a concatenation of the IV and the page offset of the block.

If one does not wish to maintain package ID, the simplest of all design choices for system ISR is to use constant IV for the entire system. This lowers the security - in this system for a given key, the plaintext-ciphertext mapping always remains constant. One could, thus, reuse code chunks with other code chunks from the same system without having to know the key. Even so, all programs that wish to run on a system have to come from a trusted source which encrypts the binaries on the system, so this may not be an acceptable security choice in many situations. The vulnerability of this scheme is same as ECB but with the performance advantages of CTR mode encryption, and design simplicity of system ISR mode.

In all the choices above, simultaneous fetch and decryption can be enabled either by storing the IV for each page in the ITLB along with the key, inferring it (in hardware) from some page characteristic, or a combination of both.

5. IMPLEMENTATION

The goal of developing the prototype was to provide a platform for full system bring-up (currently underway) as well as to understand the full impact in the context of fully fleshed out microarchitecture. In this section, we describe the changes we made in order to provide support for both modes of ISR described previously in this paper.

We used the freely-available OpenSPARC T1 microprocessor core to implement hardware support for ISR. The OpenSPARC core is supported by the Xilinx XUPV5-LX110T FPGA development board [Xilinx], which allows us to characterize the system

[3]Generating unique keys is a slightly involved process as it requires maintaining some sort of state. To prevent this complexity, the keys can be randomly selected from a large enough range so that even if all keys are not locally unique, the entropy could be large enough to offer secure diversity.

[4]Exponent obtained as $key\_width - (log_2(address\_space\_range) - log_2(cache\_width))$, assuming 128-bit key-width, 64-bit address space, and 64B cache width.

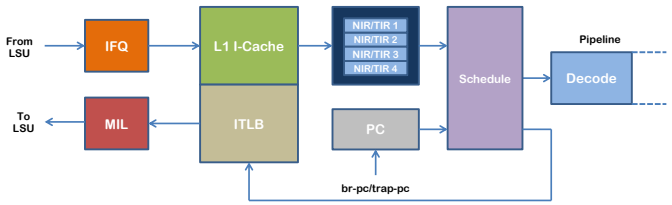

Fig. 5. Front-end of OpenSPARCT1 pipeline.

on live hardware without emulation. In our implementation, we have chosen to place the decryption module below the L1-I/memory interface because the FPGA does not have a multi-level cache. Before we describe the ISR implementation first provide a brief background on the OpenSPARC T1 microarchitecture, and then in the following section describe our hardware implementation in details.

5.1. Background on OpenSPARC T1

The Sun OpenSPARCT1 microprocessor [Sun Microsystems Inc. 2006] is an open-source variant of the Sun UltraSPARC T1 RISC architecture with its Verilog source available free. It is a 64-bit processor and supports chip multi-threading (4 per core).

The core SPARC pipeline consists of 6 stages and is part of the instruction fetch unit (IFU), which is responsible for maintaining instruction flow through the pipeline by controlling the modules feeding it. Figure 5 shows the portion of IFU at the front-end of the execution pipeline.

Four pairs of addresses are maintained for each thread, which correspond to the current PC (in the thread instruction register, TIR) and next PC (in the next instruction register, NIR) for four threads co-located at each core. Alternatively, the next address can also be determined based on an evaluated branch or interrupt. Once the next address has been determined, it is fetched from the L1-I cache which is coupled with a standard ITLB for faster address translation. Addresses that miss in the L1-I are stored in the missed instruction list (MIL) and eventually fed into the load-store unit, which is responsible for servicing misses. The retrieved instruction blocks are collected in the instruction fetch queue (IFQ) which fills the cache.

5.2. ISR implementation on OpenSPARC T1

Ideally, for single-key ISR used in ECB mode, we want the key to be stored in some non-volatile memory, from where it is loaded onto a register on boot time. This register can then be used to supply the key to the decryption unit for rest of system up-time. In our prototype, however, we hard-coded the key at design time and did not make it programmable using the test interface. This key is used to decrypt the cache lines when they are filled in from the L2 into the L1. The implementation is extremely simple. In the rest of this section, we will explore in details the various aspects of the implementation of page-mode ISR that would expect a unique key for every code page. We did not implement the other cases presented in Section 4.3, since the design would be very similar.

5.2.1. Decryption Core. The decryption core lies between the L1-I and L2 caches. It decrypts a full cache line of data as it comes in from L2, using the key previously stored in the MIL at the time of L2 request generation. This allows us to implement a more complex cipher such as AES for enhanced security. The placement of the encryption

core allows significant latency hiding, so a high-complexity algorithm will minimally effect overall system performance.

For our prototype, we implemented an AES encryption core. However, for ease of software bring-up we currently use simple combinational XOR logic between the L1-I and L2 caches as it is easier to debug.

5.2.2. Key Storage. The encryption key data (stored in the OS) is brought into the ITLB on a page fault by the miss handler of the OS or hypervisor. We use stores to alternate address spaces to perform key writes. The alternate space 0x54, already supported in hardware, is used to write ITLB physical address data. We added some alternate address space identifiers to handle key writes to the same ITLB entry. At all the places where the miss handler already uses the STXA (store eXtended word to alternate address space) instruction to write ITLB physical address data, we add similar instructions to store key data in the same entry.

5.2.3. Datapath. Instructions are decrypted as they are brought into L1-I from L2. This allows for significant latency hiding, as the majority of instruction requests are serviced from the plaintext copy in L1-I and do not require decryption.

The ITLB is augmented with a key storage memory structure that sits next to the physical address data memory. It is, hence, easy to access the keys – when a memory translation in the ITLB is accessed, the corresponding key is used to decrypt the instruction at the particular address. Keys are brought into this structure on an ITLB miss by the miss handling mechanism operated by the OS/hypervisor.

The missed instruction list (MIL) contains pending instruction requests that missed in L1-I and have been sent further out in the memory hierarchy for fulfillment. Each MIL structure is modified to add key storage, with enough space to store one full key per missed instruction. Key data is brought in from the ITLB at the same time as physical address data and the key in the corresponding MIL entry is used to decrypt the block before sending it on to the L1-I.

Furthermore, OpenSPARC conveniently uses a 128-bit L1 cache width. We, therefore, do not need to perform parallel decryption on chunks of the fetched cache block.

5.3. A Note on the Extent of Hardware Changes

Aside of the crypto accelerator that had approximately 2500 lines of Verilog code, system ISR required less than 5 lines of Verilog code of the OpenSPARC processor. Implementing page ISR design in hardware was slightly more complex. Even so, we only needed to add/modify approximately 500 lines of Verilog code (a portion of which was wire-forwarding among different modules). We have also validated our changes with extensive microbenchmarking on the prototype. Since our aim with developing a hardware prototype is to enable development of the software platform, our code changes do not include programmable ISR keys (in the single key ISR model). Even so, it is worth noting that even minimal changes in hardware code can enable very strong protections for software systems that have billions of lines of vulnerable code. We consider these to be strong arguments favoring hardware-enhanced security.

6. RESULTS

In this section, we evaluate three options for placement of the decrypting module as outlined in Section 4 and evaluate their performance overhead with respect to an unmodified machine. Our analysis shows that even using strong and expensive encryption standard such as AES for security can be done with negligible overhead.

6.1. Performance Overhead

We modified the gem5 simulator to simulate extra hit latencies corresponding to our various design points. The common architecture parameters used in the simulations is shown in Table I. The simulator itself is run in system-call emulation mode with a SimpleTiming-CPU.

Table I. Common architectural parameters.

Parameters	Specifications
Number of cores	1
L1 I-cache	32kB, 2-way, 64B line
L1 D-cache	64kB, 2-way, 64B line
L1 access latency	2 cycles
L2 cache	2MB, 8-way, 64B line
L2 access latency	20 cycles
AES latency	40 cycles
Memory access latency	60 cycles

To simulate latency corresponding to an AES decryption, we added 40 cycles (which is the latency of the AES core we implemented) to the stage at which it is be to placed. For instance, for a decryption module placed between the L1-I and the L2 caches, if the original hit latency in L2 for requests from L1-I is 100 cycles, the new hit latency now becomes 140 cycles. The miss latency is increased by an equal amount as well.

For all experiments presented here, we have used benchmarks from the SPEC CPU2006 benchmarks suite. These benchmarks were run to completion with the test input set. We chose these benchmarks because they were the only ones in the SPEC CPU2006 suite that ran without functional-emulation errors on the ALPHA/gem5 emulator in syscall mode.

Figures 6 and 7 shows the performance overhead for one of our decryption module placement choices, *viz.* at the L1-I and L2 interface. For the sake of completeness, we also examine the scheme wherein the decryption module is placed at the head of execution pipeline, *i.e.,* when every instruction fetched from the L1-I is decrypted. We do not show performance results for the case when decryption occurs between the L2 and memory, since decryption latency is much lower than memory fetch latency resulting in no performance loss, as discussed in Section 4.

We observe that placing the decryptor module before the pipeline is prohibitively expensive, whereas the other choice incurs nominal overhead.

These results are indeed what one would expect. From a high level, execution overhead will be closely correlated to how often instruction decryption occurs. Placing the decryption module at the cache interfaces results in minimal overhead because of locality at the caches (which is not the case if the module is placed above the L1). This seems to be especially true for benchmarks like gcc, gobmk and sjeng which suffer from a lot of L1-I misses. In such cases, one might consider employing our third choice, *viz.,* decrypting at the LLC-memory interface, where the performance overhead is zero irrespective of the miss profile at the LLC.

6.2. Area Overhead

We synthesized our designs for a 32nm technology-node. We found that adding an extra 16-entry 128-bit RAM (and the associated circuitry) to the ITLB for storing keys results in a 81% increase in the size of the ITLB – from 0.295 mm^2 to 0.536 mm^2. This should be manageable, however, since the actual core is orders of magnitude bigger than the ITLB.

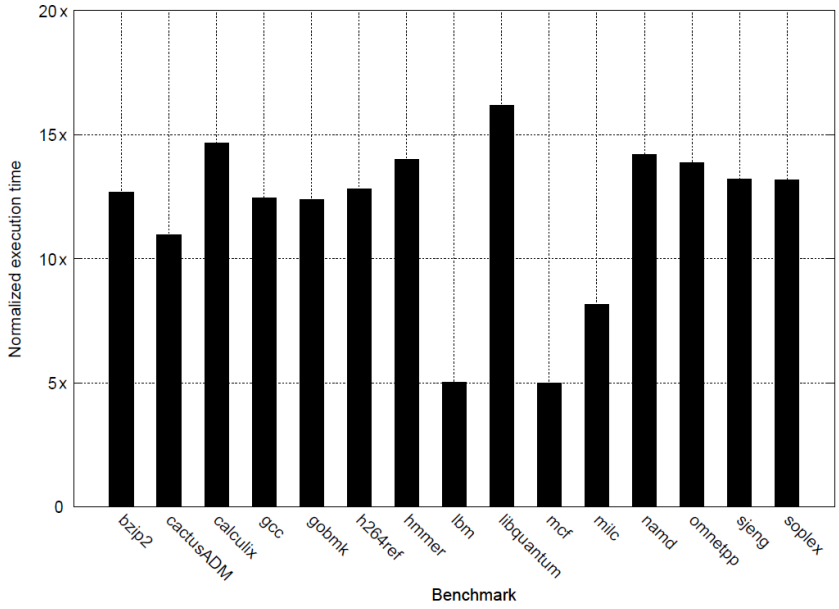

Fig. 6. Normalized execution time for each benchmark when decrypted before pipeline

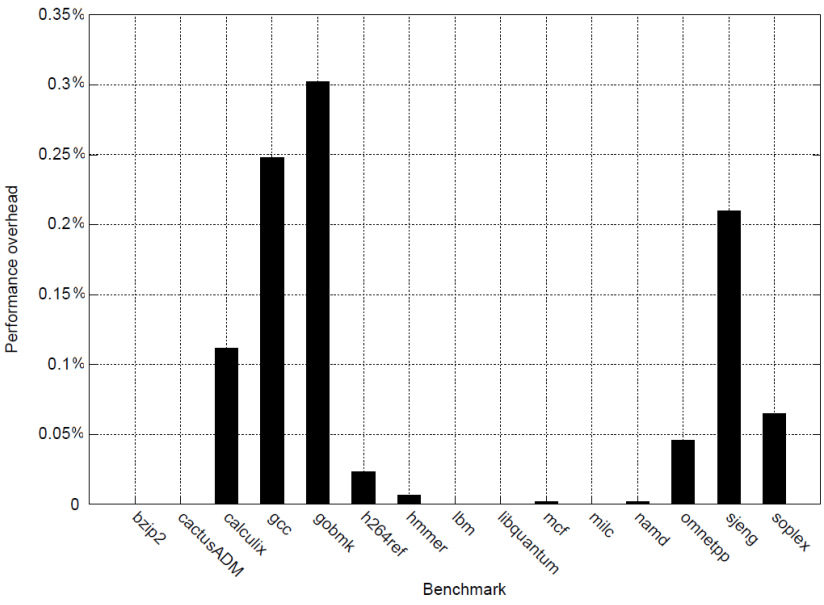

Fig. 7. Execution overhead (in %age) for each benchmark when decrypted before L1-I

We also developed a fully-pipelined AES decryptor module that takes 40 cycles to produce the 128-bit plaintext. This takes up 0.407 mm^2 of die space for the same technology.

7. SECURITY ANALYSIS

In this section, we analyze the security properties and weaknesses of our ISR scheme, according to the trust/threat model presented in Section 2.

7.1. Code-injection

Code-injection attacks are among the most widely known type of software abuse. In particular, they rely on depositing arbitrary executable code inside the address space of a victim process (typically in the form of data), and then transferring control into that code, *e.g.,* by overwriting a function pointer, a dispatch table, or a function return address. Instruction-set randomization defeats *all* types of binary code-injection attacks by design (when using a cryptographic algorithm not susceptible to known-plaintext attacks). The underlying principle behind this claim is that since an execution engine expects opcodes tailored to a particular ISA, it will not *understand* those of any other arbitrary ISA and thus fail to execute payloads comprising of such instructions. Finally, even if the valid ISA for a target were to be determined by exploiting some (local) flaw or information leak, the diversity of ISAs in the ecosystem ensures that other systems will not fall prey.

7.2. Code-reuse

Code-reuse techniques are the attackers' response to the increased adoption of system hardening mechanisms, like ASLR and NX, from commodity systems. The main idea behind code-reuse is to construct the malicious payload by reusing instructions already present in the address space of a vulnerable process. For example, if the attacker manages to diverge the control flow to the beginning of a library function such as libc's `system`, then he can spawn a shell without the need to inject any code [Designer 1997]. In most cases, however, invoking a single function is not enough for a successful compromise; multiple return-to-libc calls need to be "chained" together by reusing short instruction sequences [Nergal 2001]. ROP takes this idea to the extreme, by using only a carefully selected set of short instruction sequences, known as gadgets, for performing arbitrary computations. Note that ROP has been shown to be Turing complete, thus eliminating the need for calling libc functions [Shacham 2007].

This powerful technique gives the attacker the same level of flexibility offered by arbitrary code injection without injecting any new code at all. The malicious payload comprises just a sequence of gadget addresses intermixed with any necessary data arguments. However, in most publicly available exploits so far, attackers do not rely on a fully ROP-based payload [Pappas et al. 2012]. Typically, ROP code is used only as a first step for bypassing NX: it allocates a memory area with write and execute permissions, by calling a library function like `VirtualAlloc` or `mprotect`, copies into it some plain shellcode, and finally transfers control to the copied shellcode that now has execute permissions.

Clearly, ISR does not protect against *pure* code-reuse attacks, since the attacker can construct his payload without knowing the underlying ISA. However, there are no pure *practical* code-reuse attacks and our scheme thwarts multi-stage code-reuse exploits that rely on code injection. Finally, our scheme is orthogonal to many techniques for mitigating pure code-reuse attacks [Pappas et al. 2012; Onarlioglu et al. 2010; Kayaalp et al. 2012].

7.3. BIOS/Boot Protection

Before an OS enables privilege-protection (often with assistance from hardware), a typical system boots into a BIOS and eventually a bootloader. These two modules are very lucrative targets for attackers, since they offer virtually unrestricted access to system resources with very few, if any, monitors on their activity. Employing ISR at these stages using predetermined secret keys ensures that they cannot be patched or replaced with unverified code.

7.4. Self-Modifying Code

ISR complicates applications that employ self-modifying code (SMC), such as just-in-time (JIT) compilers. We can choose between disallowing such applications completely or selectively permitting them with a significant reduction on the security guarantees offered. Allowing SMC requires different approaches depending on the mode of ISR under consideration. For system ISR, if the SMC agent is consider part of the TCB, it could be allowed access to the system key. This significantly weakens system security by increasing the attack surface substantially and hence, is not very appealing. In case of page ISR, the SMC (or JIT) engine could either execute the generated code pages with null-keys, or encrypt it with a set of self-generated keys. Either way, some trust has to be offloaded to the SMC agent. Supporting secure JIT functionality with ISR is a topic left for future research.

7.5. Known Weaknesses

The different design choices presented in this work aim at demonstrating that ISR can indeed enable diversity across the entire system, so as to raise the bar for wide-scale exploitation. Consequently we do not claim to be immune from all types of attacks. Below we outline the weaknesses of our proposal.

Trusted OS. Page ISR requires a trusted system (*e.g.,* verified OS module) component to perform the key management for each page (per process). Unfortunately, commodity OSes are vulnerable to compromises, and are nowadays, increasingly attractive targets [Kemerlis et al. 2012].

Data-only Attacks. Since ISR does not guarantee data integrity, it is susceptible to data-only attacks (attacks that do not touch control data or modify code execution). Moreover, for applications that leak information through processed data, program state may be gleaned by observing its data.

DoS. Denial-of-Service (DoS) attacks may be carried out by repeatedly patching a program with illegal code, crashing it with "illegal instruction" exceptions. However, under most circumstances this is considered acceptable behaviour.

In spite of these drawbacks, additional security and integrity measures can be adopted on top of ISR to overcome these (or future) issues and make it more robust [Abadi et al. 2005; Parno 2008].

8. DISTRIBUTION MODELS

In any scheme like this, challenges extend beyond just diversifying at the hardware level. Practical, secure software distribution and key management also need to be engineered for wide adoption. In this section, we discuss two very different models which can be used for software distribution. One is system ISR with constant IV, and other page ISR with unique key per page. These represent two ends of the design spectrum for software distribution as we will see in this section. However, We will not go into details since secure key and content distribution are well-studied academic problems and similar models have been in practice for a long time.

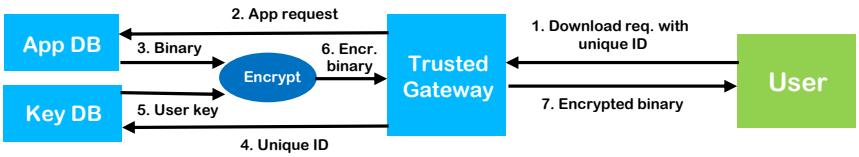

Fig. 8. Software distribution model for system ISR. User sends a download request with a unique ID (1). The app server fetches the requested binary (2, 3) and the user's system key (4,5) from back-end databases, encrypts the binary with fetched key, which is then downloaded by the user (7).

8.1. System ISR

Application Distribution Application distribution for this case is best explained in the context of the app-store model. (If we chose to use regular counter-mode, system ISR can also be implemented in the non app-store model. This would, however, look very similar to the model we describe next in Section 8.2 where asymmetric ciphers are employed to secure keys and IVs.) When a user downloads an app from the app-store, the app-store encrypts the binary with the ISR key for that device. Figure 8 illustrates this software distribution model.

(1) The chip-key as well as the unencrypted application binaries are collected and archived in trusted key and app databases respectively.
(2) When a user chooses to install an application, he sends a request to the gateway server.
(3) The gateway then sends fetch requests for the corresponding binary as well as the user's key, which is retrieved using some unique ID that was sent as part of his request.
(4) The binary is encrypted with the fetched key and sent to the user's machine.

Key Management Each processor chip is programmed with a unique random AES key. One simple way for producing chips with different keys is to include some non-volatile memory on die and then program it with a serial number and secret symmetric keys after manufacturing perhaps using the chip tester. In this scheme, we trust the chip manufacturer (or its proxy) to safely hold the keys and provide encryption service to authorized serial numbers. In the event the key is inadvertently leaked or lost, the manufacturer can (remotely) re-program the non volatile memory using secret microcode instructions (which are not ISRized). Such remote update facilities are already available in processors today to facilitate in-field correction of CPU bugs [Wagner et al. 2008].

Bootstrapping with System ISR System ISR can be turned on right from boot-up. In such a case, even the BIOS, firmware and bootloader are encrypted. No other major code or design changes need be done in these modules, a simple binary rewriting pass with the AES key is sufficient to ISRize the system. We note here that diversification does not hinder or change existing secure boot schemes where the bootloader and/or OS are verified for integrity (via comparison with some known or expected measurement of hashes on the binary image). Importantly, however, because of ISR, the measurement to be verified will not be the same across systems.

8.2. Page ISR

The previous model has two risks. Firstly, use of a single key increases risk of misuse/loss. Moreover, all applications are required to go through the app store, which might give rise to a host of undesirable problems (absence of single trusted distribution authority, less flexibility, privacy, *etc.*). Page ISR resolves these problems with a flexible key utilization system allowing use of arbitrary keys. We will describe such a

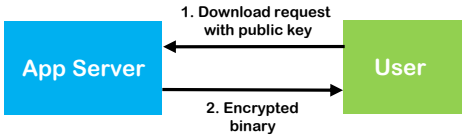

Fig. 9. Software distribution model for page ISR. Asymmetric public key is sent along with the download request (1). The app server encrypts the binary with randomly generated keys, asymmetrically the keys with the user's public key, and packages it with the binary. This binary is then downloaded by the user (2).

model below that utilizes counter-mode encryption and assumes a unique key for every code page.

Application Creation and Distribution When a developer wants to distribute an application under this system, he would be using the page-level granularity of encryption, *i.e.,* encrypt different code pages with different keys (of course, one key could correspond to more than one page).

To illustrate the sequence of events in the distribution process, let's walk through the overview presented in Figure 9.

(1) Users send download request to the app server. The public key is embedded in the request.
(2) For every download request a app server receives for his application, he encrypts the binary with a symmetric encryption algorithm like AES using developer-generated keys. These key-to-address mappings are then encrypted using the public key from user request, packaged with the binary (as an ELF section, for instance) and forwarded to the user.

Since the binary is encrypted with developer-generated keys which are in turn encrypted with the chip's public key, code confidentiality is guaranteed because the chip's private key is never revealed, not even to its owner. Moreover, even if the symmetric keys for an application were to be leaked, they would not compromise the rest of the system.

Alternatively, if code confidentiality is not necessary, applications can be downloaded in plaintext and encrypted locally.

Bootstrapping with Page-Mode ISR Page mode ISR requires paging, which is unavailable immediately on power up. The BIOS initializes the bootloader which then initializes the OS which enables paging. To enable ISR during this process we fall back to one key for the entire BIOS program with the physical load addresses acting as the counter values. The BIOS and bootloader can be prepared the same way as single key ISR applications are prepared for execution. When the OS gains control, it initializes the page tables of its own code pages with keys that were used to encrypt it. From this point onwards (*i.e.,* after virtual memory has been set up), the system starts operating in pure page ISR.

Application Execution When the program is executed, the loader extracts the encrypted key-to-page mapping from the binary and passes this information to the OS. The OS then either decrypts this mapping using the system's private key itself or asks the hardware to do so in case trusting the OS with the private key is not desirable. Once it has the decrypted mappings, it is responsible for installing the keys corresponding to each code page, both in the hardware and software address-translation mechanisms. One way to facilitate hardware translation is to extend existing page table data fields with an additional field for the 128-bit key. The OS can install the appropriate keys to each page-table entry on a page fault, which can then be transferred to the ITLB by the hardware on every ITLB miss. The decryption process uses lower

order page offset bits as the counter. This can be automatically inferred at runtime and the need to store IVs does not arise.

9. RELATED WORK

9.1. Prior Diversification Approaches

The benefits of diversification were first realized by virus writers. By creating variants, virus writers were able to bypass static signature based anti-virus schemes [Szor and Ferrie 2001]. Security researchers have also recognized it as an effective measure [Forrest et al. 1997; National Science and Technology Council 2011].

• **Address Space Layout Randomization (ASLR)** ASLR [PaX Team 2010] is a technique wherein the address space of a program is randomized to a certain degree every time it is executed. It provides diversity in terms of memory layout and is completely complementary to ISR, which diversifies code. ASLR has proven quite effective in making attacks that rely on knowledge of absolute addresses (*e.g.,* ret2libc, ROP) much harder. Another recent layout diversification technique that extends this idea to a finer granularity is that of Instruction Location Randomization [Hiser et al. 2012] wherein randomization is carried out at the instruction-level while maintaining functionality.

• **Prior ISR Implementations** Prior attempts at ISR were software-based and shown to have some security vulnerabilities [Sovarel et al. 2005]. These were then fixed, expanded in scope to cover more types of software, and optimized for speed [Portokalidis and Keromytis 2010; Boyd et al. 2010]. However, these proposals continue to have a serious flaw: in all of these proposals the randomization can be bypassed by an attacker since it is completely implemented in software. Prior proposals use dynamic rewriting tools, such as Pin, or emulators, like Bochs, to implement randomization. Obviously the attacker can turn off or subvert these components since they run at the same privilege level as the attacked application. Further, prior techniques for implementing randomization have significant slowdowns (up to 4x) making them unattractive in most environments.

By rooting diversification in hardware we cleanly sidestep these problems. First, and most important, unlike software implementations, the attacker cannot simply "turn-off" hardware rooted randomization. Second, as long as the cryptographic keys are kept secret, the attacker has a practically impossible task of figuring out the ISA of the target. Finally, using intelligent microarchitectural optimizations we are able to practically reduce the overheads of ISR to near zero.

9.2. Current Hardware-supported Protections

• **Non-executable Data Pages (NX)** The idea of non-executable page was proposed [Designer 1997] in software more than a decade ago, and Intel, AMD and ARM have begun supporting it in hardware. Although NX by itself can be defeated easily, when used in conjunction with other diversification techniques like ASLR, the bar for attacks is raised much higher. This is a great example of how a combination of piecewise insecure techniques can combine to strengthen overall system defense.

• **Code Signing and Trusted Platform Module (TPM)** Code signing is a technique used to guarantee the authenticity and integrity of applications. During application installation [Apple Inc.] or launch [Parno et al. 2010] a cryptographic hash on the binary is checked with a "golden" trusted value. This verification is done on static non-executing code. ISR, on the other hand, operates on code in motion *i.e.,* at runtime. TPMs [tpm 2003] basically provide secure storage of the "golden" values for purposes like code signing. They are completely orthogonal to ISR.

9.3. Crypto-microprocessors

The idea of microprocessors supporting encrypted execution were proposed more than three decades ago [Best 1981] and has been revisited regularly in various forms ever since [White and Comerford 1990; Lie et al. 2000; Suh et al. 2003b; Lee et al. 2005; Williams and Boivie 2011]. Most of earlier work were designed with different goals in keeping with contemporary trends and were not designed for diversity, or for the current cloud-mobile computing model. Consequently, key design determiners such as threat model, software distribution model and hardware design possibilities are quite different: for instance, the software distribution model for one of our diversification schemes would be impractical in the pre-mobile era.

We specifically focus on more recent work focusing on providing secure execution, *viz.,* XOM [Lie et al. 2000], AEGIS [Suh et al. 2003b], SP [Lee et al. 2005] and Secure Executable (SE) [Williams and Boivie 2011]. They aim to provide hardware support for integrity and confidentiality of both code and data, even in the presence of a malicious operating system. The idea is to achieve security by obfuscating code and data outside the perimeters of a secure hardware (typically the processor is trusted) while providing some isolation within. Each of these suffer from one or more of the following drawbacks.

— **Support for shared libraries** They enforce very strong security constraints and isolation among untrusted entities. Hence, providing support, if at all, for shared libraries is quite involved. ISR, on the other hand, provides seamless support for shared libraries.
— **Notion of compartmentalized secure execution** This line of work requires changes in code and often in design of software in order to determine what chunks of code are security-critical. Basically this means that their security benefits do not extend to legacy code. ISR, however, is completely transparent to developers with the exception of self-modifying code and JIT compilers which requires some programmer support (See Section 7.4).
— **Can be turned off** The above problems could be mitigated if one chose to run entire applications in secure mode (as does SE, for instance). However, this may not be enough since the security mechanisms can be turned off. Even with all the protections, these systems are still vulnerable to attacks (*e.g.,* control flow manipulation via malformed inputs), some of which may be enabled by actual bugs in the code itself. An attacker only needs to execute the "exit secure-mode" instruction somehow and all protections would be repealed. In ISR we do not have a non-ISR mode of operation. Our scheme is always “on”. We can have ISR mode on essentially from the very first instruction executed on a CPU cold-boot and throughout its operation (which is why low overheads are critical).

Additionally, none of them offer a full end-to-end analysis of changes required in the ecosystem required to accommodate their respective implementations, which is a non-trivial aspect of any proposal of this nature.

Counter mode encryption for code and data memory has also received a lot of attention academically [Suh et al. 2003a; Shi et al. 2005; Yan et al. 2006] mainly because it allows random access as well as parallel decryption and fetch. Although some of them have also proposed using some notion of address and the ITLB to facilitate decryption, other structures such as caches and predictors are recommended as well . Furthermore, in our scheme, IVs are completely known right when a fetch has missed because we infer it in its entirety from the address and for some designs, additional ITLB entries.

10. CONCLUSION

Once a security flaw is discovered, attacks exploiting it can be quickly and widely deployed due to the homogeneity of computing ecosystem. Diversity is a highly desirable feature to limit the damage such security lapses incur. In this paper we present a solution for randomizing instruction sets that is highly secure and has no performance overheads.

We implemented this technique by encrypting the instruction stream with AES and decrypting it before execution. Additionally, the encryption key for AES is kept secret to prevent development of unauthorized binary. We have also developed microarchitectural optimizations to perform this with zero overhead. A particularly striking feature of the ISR solution is that very simple modifications in hardware enable broad security coverage; for instance, we protect against all kinds of binary code-injection attacks.

We also explored some ideas as to how entire ecosystems could be developed around the different types of proposed ISR. An advantage of these diversification approaches based around ISR is that they employ practical cryptographic primitives and can be built on top of established models. Moreover, they are orthogonal to and can be used alongside most other code-signing and security measures to further harden systems.

ISR provides a foundation for further security improvements. One such direction is offloading key-management to hardware so that the requirement of a trusted OS can be done away with, thus evicting thousands of lines of software code from the sphere of trust. Yet another is to extend this encryption on data items which can open the way for truly obfuscated and practical "dark execution" which is particularly significant today when executing programs on remote clouds. Thus hardware ISR is a promising security primitive with many benefits.

REFERENCES

2003. Trusted Computing Group. http://www.trustedcomputinggroup.org. (2003).

Apple Inc. . Apple Code Signing Guide. https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html. (????).

Martín Abadi, Mihai Budiu, Úlfar Erlingsson, and Jay Ligatti. 2005. Control-flow integrity. In *Proceedings of the 12th ACM conference on Computer and communications security (CCS '05)*. ACM, New York, NY, USA, 340–353. DOI: http://dx.doi.org/10.1145/1102120.1102165

Ahmed M. Azab, Peng Ning, and Xiaolan Zhang. 2011. SICE: a hardware-level strongly isolated computing environment for x86 multi-core platforms. In *Proceedings of the 18th ACM conference on Computer and communications security (CCS '11)*. ACM, New York, NY, USA, 375–388. DOI: http://dx.doi.org/10.1145/2046707.2046752

Robert M. Best. 1981. Crypto microprocessor for executing enciphered programs. (04 1981).

H. Binsalleeh, T. Ormerod, A. Boukhtouta, P. Sinha, A. Youssef, M. Debbabi, and L. Wang. 2010. On the Analysis of the Zeus Botnet Crimeware Toolkit. In *Proceedings of the 8th Annual International Conference on Privacy Security and Trust (PST)*. Ottawa, Ontario, Canada, 31–38.

Stephen W. Boyd, Gaurav S. Kc, Michael E. Locasto, Angelos D. Keromytis, and Vassilis Prevelakis. 2010. On the General Applicability of Instruction-Set Randomization. *IEEE Trans. Dependable Secur. Comput.* 7, 3 (July 2010), 255–270. DOI: http://dx.doi.org/10.1109/TDSC.2008.58

Solar Designer. 1997. Getting around non-executable stack (and fix). http://seclists.org/bugtraq/1997/Aug/63. (August 1997).

Morris Dworkin. 2001. Recommendations for Block Cipher Modes of Operation: Methods and Techniques. http://csrc.nist.gov/publications/nistpubs/800-38a/sp800-38a.pdf. (2001).

Stephanie Forrest, Anil Somayaji, and David H. Ackley. 1997. Building Diverse Computer Systems. In *Workshop on Hot Topics in Operating Systems*. 67–72.

Jason Hiser, Anh Nguyen-Tuong, Michele Co, Matthew Hall, and Jack W. Davidson. 2012. ILR: Where'd My Gadgets Go?. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy (SP '12)*. IEEE Computer Society, Washington, DC, USA, 571–585. DOI: http://dx.doi.org/10.1109/SP.2012.39

Mehmet Kayaalp, Meltem Ozsoy, Nael Abu-Ghazaleh, and Dmitry Ponomarev. 2012. Branch regulation: low-overhead protection from code reuse attacks. In *Proceedings of the 39th Annual International Sym-*

posium on Computer Architecture (ISCA '12). IEEE Computer Society, Washington, DC, USA, 94–105. http://dl.acm.org/citation.cfm?id=2337159.2337171

Vasileios P. Kemerlis, Georgios Portokalidis, and Angelos D. Keromytis. 2012. kGuard: Lightweight Kernel Protection against Return-to-user Attacks. In *Proceedings of the* 21^{st} *USENIX Security Symposium (USENIX Sec)*. 459–474.

Lavasoft Corporation. 2006. A Closer Look at Zlob Trojans. http://www.lavasoft.com/mylavasoft/securitycenter/articles/zlobtrojans. (2006).

Ruby B. Lee, Peter C. S. Kwan, John P. McGregor, Jeffrey Dwoskin, and Zhenghong Wang. 2005. Architecture for Protecting Critical Secrets in Microprocessors. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 2–13. DOI: http://dx.doi.org/10.1109/ISCA.2005.14

D. Lie, C. Thekkath, P. Lincoln, M. Mitchell, D. Boneh, J. Mitchell, and M. Horowitz. 2000. Architectural Support for Copy and Tamper Resistant Software. In *Architectural Support for Programming Languages and Operating Systems (ASPLOS-IX)*. Cambridge, MA. http://www.csl.sri.com/papers/lincolnxom2000/

MITRE Corporation. 2012. Common Weakness Enumeration. http://cwe.mitre.org/data/definitions/94.html. (November 2012).

National Science and Technology Council. 2011. Trustworthy Cyberspace: Strategic plan for the Federal Cybersecurity Research and Development Program. (December 2011).

Nergal. 2001. The advanced return-into-lib(c) exploits: PaX case study. http://www.phrack.org/issues.html?issue=58&id=4#article, *Phrack* 11, 58 (December 2001).

National Institute of Standards and Technology. 2001. *FIBS 197, Advanced Encryption Standard (AES)*. Technical Report.

Kaan Onarlioglu, Leyla Bilge, Andrea Lanzi, Davide Balzarotti, and Engin Kirda. 2010. G-Free: Defeating Return-Oriented Programming through Gadget-less Binaries. In *Proceedings of the* 26^{th} *Annual Computer Security Applications Conference (ACSAC)*. 49–58.

Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proceedings of the 2012 IEEE Symposium on Security and Privacy*. San Fransisco, CA, USA, 601–615.

Bryan Parno. 2008. Bootstrapping trust in a "trusted" platform. In *Proceedings of the 3rd conference on Hot topics in security (HOTSEC'08)*. USENIX Association, Berkeley, CA, USA, Article 9, 6 pages.

Bryan Parno, Jonathan M. Mccune, and Adrian Perrig. 2010. Bootstrapping trust in commodity computers. In *In Proceedings of the IEEE Symposium on Security and Privacy*.

PaX Team. 2010. PaX address space layout randomization. (2010). http://pax.grsecurity.net/docs/aslr.txt

Georgios Portokalidis and Angelos D. Keromytis. 2010. Fast and practical instruction-set randomization for commodity systems. In *Proceedings of the 26th Annual Computer Security Applications Conference (ACSAC '10)*. ACM, New York, NY, USA, 41–48. DOI: http://dx.doi.org/10.1145/1920261.1920268

Niels Provos, Dean McNamee, Panayiotis Mavrommatis, Ke Wang, and Nagendra Modadugu. 2007. The Ghost in the Browser Analysis of Web-based Malware. In *Proceedings of the* $1st$ *Workshop on Hot Topics in Understanding Botnets (HotBots)*. Cambridge, MA, USA.

Anibal L. Sacco and Alfredo A. Ortega. 2009. Persistent BIOS Infection. http://www.phrack.com/issues.html?issue=66&id=7, *Phrack* 66 (June 2009).

Hovav Shacham. 2007. The geometry of innocent flesh on the bone: return-into-libc without function calls (on the x86). In *Proceedings of the 14th ACM conference on Computer and communications security (CCS '07)*. ACM, New York, NY, USA, 552–561. DOI: http://dx.doi.org/10.1145/1315245.1315313

Weidong Shi, Hsien-Hsin S. Lee, Mrinmoy Ghosh, Chenghuai Lu, and Alexandra Boldyreva. 2005. High Efficiency Counter Mode Security Architecture via Prediction and Precomputation. In *Proceedings of the 32nd annual international symposium on Computer Architecture (ISCA '05)*. IEEE Computer Society, Washington, DC, USA, 14–24. DOI: http://dx.doi.org/10.1109/ISCA.2005.30

Ana Nora Sovarel, David Evans, and Nathanael Paul. 2005. Where's the FEEB? The effectiveness of instruction set randomization. In *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*. USENIX Association, Berkeley, CA, USA, 10–10.

Brad Spengler. 2007. On exploiting null ptr derefs, disabling SELinux, and silently fixed Linux vulns. http://seclists.org/dailydave/2007/q1/224. (March 2007).

Brett Stone-Gross, Ryan Abman, RichardA. Kemmerer, Christopher Kruegel, DouglasG. Steigerwald, and Giovanni Vigna. 2013. The Underground Economy of Fake Antivirus Software. In *Economics of Information Security and Privacy III*, Bruce Schneier (Ed.). Springer New York, 55–78. DOI: http://dx.doi.org/10.1007/978-1-4614-1981-5_4

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003a. Efficient Memory Integrity Verification and Encryption for Secure Processors. In *Proceedings of the 36th annual IEEE/ACM International Symposium on Microarchitecture (MICRO 36)*. IEEE Computer Society, Washington, DC, USA, 339–. http://dl.acm.org/citation.cfm?id=956417.956575

G. Edward Suh, Dwaine Clarke, Blaise Gassend, Marten van Dijk, and Srinivas Devadas. 2003b. AEGIS: architecture for tamper-evident and tamper-resistant processing. In *Proceedings of the 17th annual international conference on Supercomputing (ICS '03)*. ACM, New York, NY, USA, 160–171. DOI: http://dx.doi.org/10.1145/782814.782838

Sun Microsystems Inc. 2006. OpenSPARC T1 Micro Architecture Specification. (2006).

Symantec Corporation. 2012. Symantec Internet Security Threat Report. (April 2012). http://www.symantec.com/threatreport/

Peter Szor and Peter Ferrie. 2001. Hunting for metamorphic. In *In Virus Bulletin Conference*. 123–144.

Trend Micro Corporation. 2012. Russian Underground. http://www.trendmicro.com/cloud-content/us/pdfs/security-intelligence/white-papers/wp-russian-underground-101.pdf. (October 2012).

Ilya Wagner, Valeria Bertacco, and Todd Austin. 2008. Using Field-Repairable Control Logic to Correct Design Errors in Microprocessors. *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 27, 2 (Feb. 2008), 380–393. DOI: http://dx.doi.org/10.1109/TCAD.2007.907239

Steve R. White and Liam Comerford. 1990. ABYSS: An Architecture for Software Protection. *IEEE Trans. Softw. Eng.* 16, 6 (June 1990), 619–629. DOI: http://dx.doi.org/10.1109/32.55090

Peter Williams and Rick Boivie. 2011. CPU support for secure executables. In *Proceedings of the 4th international conference on Trust and trustworthy computing (TRUST'11)*. Springer-Verlag, Berlin, Heidelberg, 172–187.

Rafal Wojtczuk and Alexander Tereshkin. 2009. Attacking Intel BIOS. In *Black Hat USA*. Las Vegas, Nevada, USA.

Xilinx. Xilinx University Program XUPV5-LX110T Development System. (????). http://www.xilinx.com/univ/xupv5-lx110t.htm

Chenyu Yan, Daniel Englender, Milos Prvulovic, Brian Rogers, and Yan Solihin. 2006. Improving Cost, Performance, and Security of Memory Encryption and Authentication. In *Proceedings of the 33rd annual international symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, Washington, DC, USA, 179–190. DOI: http://dx.doi.org/10.1109/ISCA.2006.22