Thwarting Cache Side-Channel Attacks Through Dynamic Software Diversity

Stephen Crane, Andrei Homescu, Stefan Brunthaler, Per Larsen, and Michael Franz
University of California, Irvine
{sjcrane, ahomescu, s.brunthaler, perl, franz}@uci.edu

***Abstract*—We explore software diversity as a defense against side-channel attacks by dynamically and systematically randomizing the control flow of programs. Existing software diversity techniques transform each program trace *identically*. Our diversity based technique instead transforms programs to make each program trace *unique*. This approach offers probabilistic protection against both online and off-line side-channel attacks.**

In particular, we create a large number of unique program execution paths by automatically generating diversified replicas for parts of an input program. Replicas derived from the same original program fragment have different implementations, but perform semantically equivalent computations. At runtime we then randomly and frequently switch between these replicas.

We evaluate how well our approach thwarts cache-based side-channel attacks, in which an attacker strives to recover cryptographic keys by analyzing side-effects of program execution. Our method requires no manual effort or hardware changes, has a reasonable performance impact, and reduces side-channel information leakage significantly.

I. Motivation

Artificial software diversity, like its biological counterpart, is a highly flexible and efficient defense mechanism. Code injection, code reuse, and reverse engineering attacks are all significantly harder against diversified software ([1], [2], [3], [4], [5], [6], [7], [8]). We propose to extend software diversity to protect against side-channel attacks, in particular cache side channels.

Essentially, artificial software diversity denies attackers precise knowledge of their target by randomizing implementation features of a program. Because code reuse and other related attacks rely on *static properties* of a program, previous work on software diversity predominantly focuses on randomizing the program *representation*, e.g., the in-memory addresses of code and data. Side-channel attacks, on the other hand, rely on *dynamic properties* of programs, e.g., execution time, memory latencies, or power consumption. Consequently, diversification against side channels must randomize a program's execution rather than its representation.

Permission to freely reproduce all or part of this paper for noncommercial purposes is granted provided that copies bear this notice and the full citation on the first page. Reproduction for commercial purposes is strictly prohibited without the prior written consent of the Internet Society, the first-named author (for reproduction of an entire paper only), and the author's employer if the paper was prepared within the scope of employment.
NDSS '15, 8-11 February 2015, San Diego, CA, USA
Copyright 2015 Internet Society, ISBN 1-891562-38-X
http://dx.doi.org/10.14722/ndss.2015.23264

Most existing diversification approaches randomize programs before execution, e.g., during compilation, installation, or loading. Ahead-of-time randomization is desirable because re-diversification during runtime impacts performance (similar to just-in-time compilation). Some approaches interleave program randomization and program execution ([9], [6], [10], [11]). However, the granularity of randomization in these approaches is quite coarse, potentially allowing an attacker to observe the program uninterrupted for long enough to carry out a successful side-channel attack. We avoid this problem by extending techniques used to prevent reverse engineering such as code replication and control-flow randomization ([12], [7]). Unlike these approaches, however, we replicate code at a finer grained level and produce a nearly unlimited number of runtime paths by randomly switching between these replicas. Rather than making control flow difficult to reverse engineer, our technique randomly switches execution between different copies of program fragments, which we refer to as replicas, to randomize executed code and thus side-channel observations. We call this new capability *dynamic control-flow diversity*.

To vary the side-channel characteristics of replicas, we employ diversifying transformations. Diversification preserves the original program semantics while ensuring that each replica differs at the level of machine instructions. To protect against cache side-channel attacks we use diversifications that vary observable execution characteristics. Like other cache side-channel mitigations, such as reloading the cache on context switches and rewriting encryption routines to avoid optimized lookup tables, introducing diversity has some performance impact which we rigorously quantify in this paper.

In combination, dynamic control-flow diversity and diversifying transformations create binaries with randomized program traces, without requiring hardware or developer assistance. In this paper we explore the use of dynamic control-flow diversity against cache-based side-channel attacks on cryptographic algorithms. Our main contributions are the following:

- We apply the new capability of dynamic control-flow diversity to the problem of side channels. To the best of our knowledge, this is the first use of automated software diversity to mitigate cache side channels.
- We show how to generate machine code for efficient randomized control-flow transfers and combine this with a diversifying transformation to counter cache-based side-channel attacks.
- We present a careful and detailed evaluation of applying diversity to protect cache side channels and report the following:

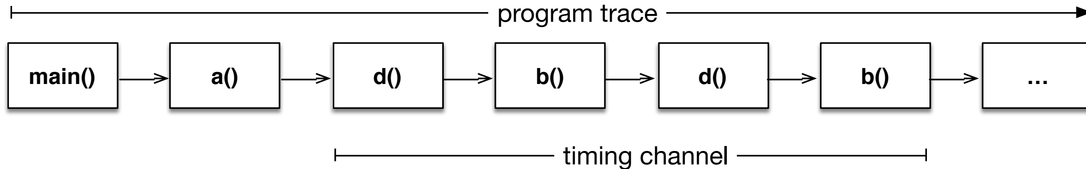

Fig. 1: Time based side channel exploitable through a sequence of function calls in a program trace.

- **Security**: Our techniques successfully mitigate two realistic cache side-channel attacks against AES on modern hardware.
- **Performance**: Applying dynamic control-flow diversity with effective security settings to an AES micro-benchmark of the libgcrypt library results in performance impacts of 1.75x and protecting a real-world application using AES results in a slowdown of 1.5x.

II. Side-Channel Background

The execution of a program is described by its control flow. The sequence of all control-flow transitions a program takes during execution is usually referred to as an execution path, or a *program trace*. A program trace describes the dynamic behavior of a program. Figure 1 illustrates a program trace at the granularity of function calls.

Executing programs on real hardware results in dynamic properties that leak information, such as timing or power variation. For example, Figure 1 shows a side channel based on time spent in executing the function sequence d(), b(), d(), b(). By observing dynamic properties of a program trace through a side channel, attackers can derive information about the actual program execution, such as inferring secret inputs to the program.

A. Threat Model

Since side-channel attacks often target secret keys of a process performing encryption, in this paper we assume that an attacker is targeting such a secret key. To demonstrate the applicability of our techniques, we assume an advantageous scenario for this attacker and reason that our defense remains effective under weaker assumptions.

Tromer et al. [13] classified side-channel attacks into synchronous and asynchronous attacks depending on whether or not the attacker can trigger processing of known inputs (usually plain- or cipher-texts). Synchronous encryption attacks, where the attacker can trigger and observe encryption of known messages, are generally easier to perform, and thus harder to defend against, since the attack does not need to determine the start and end of each encryption. We assume as strong a position for the attacker as possible and therefore will consider the scenario where an attacker can request and observe encryption of arbitrary chosen plaintexts.

To minimize external noise, we assume that the attacker is co-resident on the same machine as the target process. We also assume that the attacker can execute arbitrary user-mode code on a processor core shared with the target process but does not have access to the address space of the target process.

In the interest of allowing a strong attacker model, we advise but do not require that the protected binary be kept secret. Since we randomly generate diverse but semantically equivalent binaries, preventing the attacker from reconstructing the target environment is an advisable defense-in-depth against off-line attacks, such as the cross-VM attack described by Zhang et al. [14]. Deploying protected programs with differing layouts is also an effective defense against code-reuse attacks [15] and we can defend in the same manner by deploying randomized binaries which include dynamic control-flow diversity.

If we allow access to the binary, we must be careful that the attacker is not able to accurately determine which replica of each program unit was executed in an observed program trace. An attacker who observes a complete trace of control-flow transfers could filter out the effects of the replicas' diversifying transformations, regardless of what those effects are. In practice, a user-level process cannot observe all control-flow transfers of another process, especially at the granularity of basic blocks[1].

B. Example Attacks

To demonstrate an example of our dynamic control-flow diversity defense, we chose two synchronous, known input cache attacks on AES described by Tromer et al. [13]: EVICT+TIME and PRIME+PROBE. Although these representative cache attacks have limited scope, an attacker could use this type of attack to compromise a system-wide filesystem encryption key or target a proxy server where an attacker can trigger encryption of known plaintexts. In addition, these attacks are representative of cache-based side channels and are the basis of several more complex side-channel attacks [14], [17], [18]. While we demonstrate the effectiveness of our technique against cache-based side channels in particular, we expect that the same general defense paradigm can be applied to other categories of side channels using different diversifying transformations than the ones we discuss in Section III-A.

Caches exploit temporal and spatial locality to speed up access to recently used data. This helps to compensate for the speed gap between processors and main memories. As a side effect, caches increase the correlation between program inputs and its execution characteristics.

Modern processors access the cache in units called "cache lines," which are typically 64 bytes long. Each cache level is partitioned into n "cache sets," and each memory line can

[1]Gullasch et al. [16] describe a DoS attack against the OS scheduler which could result in such fine-grained information, but the OS scheduler can be hardened to prevent such attacks.

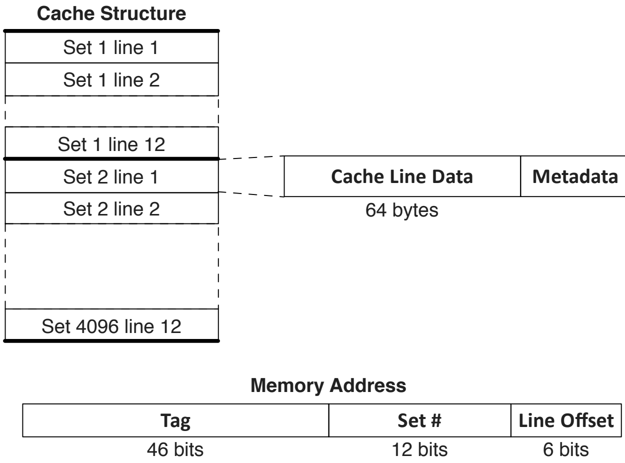

Fig. 2: Example of cache structure on a modern processor. Cache shown is 3MB in size, with 4096 (2^{12}) sets, 12-way associativity and 64-byte cache lines. Memory addresses are broken into a 46-bit (or less) tag, a 12-bit set number and a 6-bit line offset.

be placed into exactly one of these n sets. Each set stores at most m lines simultaneously, in which case the cache is called "m-way set associative". In practice, caches are 4-, 8-, 12- and 16-way associative. Figure 2 shows the structure of a 3MB 12-way set associative cache found in our test system.

For efficiency, the processor shares these caches between running processes but prevents processes from accessing data belonging to other processes via the virtual memory abstraction. However, since data from multiple processes is concurrently stored in the cache, adversaries can indirectly deduce information about which cache locations a target process accesses by observing side-effects of cache accesses. Since the data cache access patterns of many programs are input-dependent and predictable, attackers can use knowledge of some inputs and the target's data access patterns to derive the secret input.

To exploit cache access patterns, all cache timing attacks rely on the same fundamental principle of cache behavior: accessing data stored in the cache is measurably faster than accessing the data from main memory. As a result, attacks can exploit this principle as a side channel and observe different cache behavior for certain segments of a program trace. In the EVICT+TIME attack, we observe the effect of evicting an entire cache set and forcing the encryption program to fetch values from main memory, while in the PRIME+PROBE attack we fill a cache set and check which cache lines the encryption evicts by observing the time to reload our data.

For convenience we summarize both AES attacks here but refer interested readers to Tromer et al. [13] for further details. Optimized AES implementations use four in-memory tables (T_0 through T_3, each containing 256 four-byte values) during encryption, and the access pattern of these tables varies according to the key and plaintext inputs. Specifically, during the first of ten encryption rounds for plaintext $\mathbf{p}$ and key k, the encryption process will access table T_l at index $p_i \oplus k_i$

Input : Cache set c to probe, plaintext p, key k.
Output: Time needed to encrypt the plaintext after probing c.

```
Encrypt(k, p);
Evict cache set c;
t0 ←Time();
Encrypt(k, p);
t1 ←Time();
return t1 − t0;
```

Algorithm 1: EVICT+TIME attack.

Input : Array C of cache sets to probe, plaintext p, key k.
Output: Array T of times needed to probe each set in C.

```
foreach c ∈ C do
    Read w values into cache set c from memory;
end
Encrypt(k, p);
foreach c ∈ C do
    t0 ←Time();
    Read w values from cache set c;
    t1 ←Time();
    T[c] ← t1 − t0;
end
return T;
```

Algorithm 2: PRIME+PROBE attack.

for all $i = 0, \ldots, 15$ where $l = i \bmod 4$. Since we assume the attacker knows the plaintext $\mathbf{p}$, the attacker is able to derive information about the key from information about which table elements are loaded from memory.

Algorithm 1 shows the EVICT+TIME attack. We derive the table access patterns by observing the total execution time of the encryption routine. By first running the encryption on a chosen, random plaintext, we prime the cache with the table entries required during the encryption of this plaintext. We then completely evict a cache set by loading a set of memory locations that all map into the chosen cache set. By timing another encryption of the same plaintext, we can then, by averaging over many runs, determine whether the encryption used a table value from that cache set, since the encryption routine will take longer when accessing an evicted table entry due to the cache miss.

The PRIME+PROBE attack (shown in Algorithm 2) is very similar to the EVICT+TIME attack, but with the timing and eviction roles flipped. In this attack we first create a known starting cache state by loading a set of memory locations into each relevant cache set. We then trigger encryption of a chosen plaintext, which will modify this cache state by caching accessed table entries. Finally, we determine which cache sets were modified by timing a load of each cache set again. The cache sets corresponding to table entries that the encryption accessed will take longer to load than those not used, since the encryption table entry will have displaced one of the original entries loaded by the attacker and thus incur at least one cache miss.

By analyzing a large set of these cache observations for randomly chosen plaintexts, we can determine the key bits that

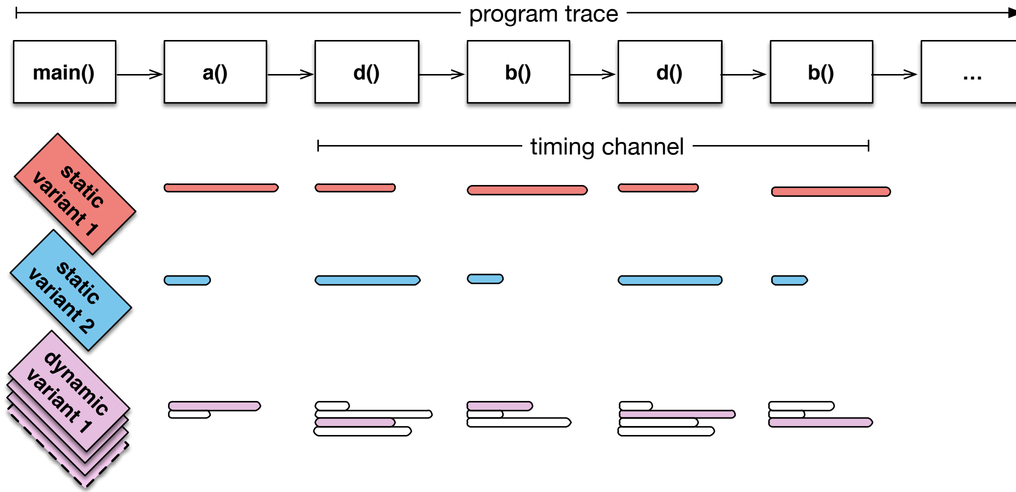

Fig. 3: Side-channel resistance of diversification techniques.

correspond to table indices in the first round of encryption. For each guess of a key byte $\hat{k}_i$, we average all observed timings for the cache set evictions corresponding to table entry $T_{i \bmod 4}[\hat{k}_i \oplus p_i]$. In both attacks, the highest observed average time should correspond to the correctly guessed key byte. However, with 64 byte cache lines, four table entries fit into each cache line, and we can only observe accesses at the granularity of cache lines, which means that we can only determine the high nibble of each key byte with this analysis. Therefore, to determine the lower four bits of each key byte, we must analyze the second round of encryption as described by Tromer et al. This analysis, while more involved, is conceptually analogous to the first round analysis and we refer interested readers to the description in the original paper.

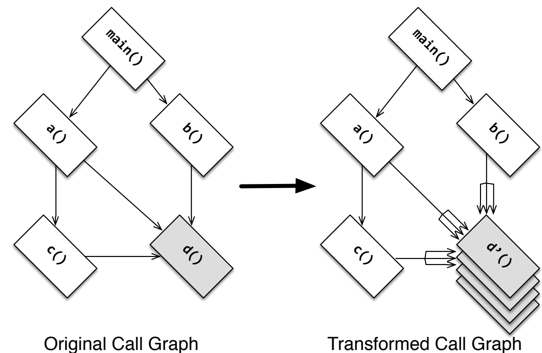

Fig. 4: Function call graph before and after replicating the function d().

III. Dynamic Control-Flow Diversity

Most diversification techniques prevent attackers from constructing reliable attacks by randomizing the layout of a program's data and code. Since modern exploits such as code reuse attacks depend on detailed knowledge of the program layout and internals, automatically modifying these aspects of the program implementation hinders development of reliable exploits using techniques such as return-oriented programming. However, software diversity affects not only program layout but also alters program side-effects, such as run time, power usage and cache usage. Even simply re-ordering functions can have a large effect on cache usage and performance since code will be aligned differently in the instruction cache.

Since software diversity affects performance and cache usage, by extension we observed that it could be useful to disrupt or add noise to side channels. However static compile-time or load-time diversity is insufficient, since side-channel attacks are online dynamic attacks and attackers can simply profile the static target binary to learn its runtime characteristics. Re-diversifying and switching to a new variant during execution is also insufficient since side-channel attacks are fast enough to complete between reasonably spaced re-diversification cycles. Figure 3 illustrates the effect of diversification techniques on side channels. While the program trace of the original program leaves a specific footprint on the executing hardware, diversified program variants (labeled as static variant 1 and static variant 2) each have a different footprint. This diversity is likely to thwart offline profiling attacks, but online side-channel attacks that deduce information by monitoring the running program are not affected by these diversification techniques.

We extend previous, mostly static software diversification approaches by dynamically randomizing the control flow of the program while it is running. Rather than statically executing a single variant each time a program unit is executed, we create a program consisting of replicated code fragments with randomized control flow to switch between alternative code replicas at runtime.

In Figure 3, we see the effect of dynamic control-flow diversity in the bottom row, labeled dynamic variant 1. For the trace segment the attacker is interested in, the program can now take numerous different paths, effectively preventing the attacker from constructing a reliable model to infer program execution information from side-channel characteristics, such as timing.

We build our control-flow diversity on a conventional compiler-based diversification system that creates random-

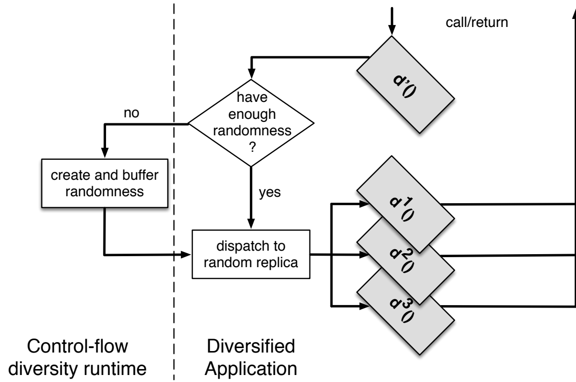

Fig. 5: Detailed view of randomized trampoline d'() and interaction with the runtime system.

ized variants of a program fragment, such as a function or a basic block, by applying diversifying transformations. A diversifying transformation preserves program semantics but transforms implementation details. Examples of previously proposed diversifying transformations include insertion of NOP instructions, permutation of function or basic block layout, and randomization of register assignments. In Section III-A we discuss a diversifying transformation to illustrate the effects of control-flow diversity on cache side channels, but other transformations could be used to protect against other instances and varieties of side-channel attacks.

To create control-flow diversity, we begin by choosing a set of program fragments (either functions or basic blocks) to transform. If a developer knows that some sections of the program, such as encryption routines, are particularly interesting targets for side-channel attacks, the developer can manually specify this set of program fragments to diversify. In addition, for blanket coverage we can randomly select candidate program fragments. Since randomized control-flow transfers add performance overhead, the software distributor should adjust the percentage of duplicated fragments to balance security and performance.

After choosing a set of functions and/or basic blocks, we clone each chosen program fragment a configurable number of times. We then use different diversifying transformations for each clone to create functionally-equivalent replicas that differ in runtime characteristics. The set of transformations applied to each program fragment may include completely different transformations, applications of the same transformation with different parameters, or some combination of both. Figure 4 shows an example of this process applied to a function.

We then integrate these randomized replicas into a program that dynamically chooses control-flow paths at runtime. For each replica, we replace all references to the original fragment with a *randomized trampoline*. As illustrated in Figure 5, whenever the program executes a trampoline it randomly chooses a replica to transfer control to.

We use the SIMD-Oriented Fast Mersenne Twister pseudo-random number generator (PRNG) [19], since the runtime needs to quickly generate random numbers. Although our chosen PRNG is not cryptographically secure, it is sufficient for our purposes, since we assume the attacker cannot extract every control-flow transfer through the noisy side channel. If defending a side channel through which extracting the dynamic control flow and predicting the PRNG stream is easier than extracting the targeted secret information, this PRNG could easily be replaced by a cryptographically secure PRNG. Processor-integrated random number generators would be ideal to fill this role, and, as processors with this capability become widespread, we expect that the processor can fill a randomness buffer instead of using a software PRNG.

A. Cache Noise Transformation

In order to produce structurally different but semantically identical variants, we randomly apply diversifying transformations to the program code. These transformations change how a program looks to an observer (who might either read the binary itself, or observe it through side channels) without affecting program semantics. We investigated one specific transformation, inserting cache noise, to disrupt cache side-channel observations. However, this technique is only one example of possible side-channel disrupting transformations. When protecting other side channels, one may need different transformations, e.g., disrupting power observations might require randomly weaving in another unrelated program to ensure that the inserted code is indistinguishable from the original program code.

We initially investigated disrupting the EVICT+TIME attack by randomly inserting NOP instructions into the code. However, after optimizing our randomness generation, we found that NOP instructions do not add enough time fluctuation to disrupt the attack. In addition, NOP instructions have no effect on cache usage, and thus do nothing to affect the PRIME+PROBE attack. We therefore turned our attention to inserting random memory loads, which disrupt both timing and cache snooping side channels.

To ensure that inserted loads have a high likelihood of actually impacting the performance of the targeted program, we want to create loads that evict a specific set of cache lines, specifically those that the target uses. In addition, attempting to read from invalid addresses (such as unallocated regions in the process address space) can potentially crash the target program, stopping the attack. For these reasons, we restrict the loads to a linear region, selected at program load-time. In the case of our AES experiments, this region covers only the AES S-box tables but in general is adjustable for other applications.

Our compiler randomly picks the locations to insert loads at compile time, and the target program itself picks the base and size of the region at load-time during program initialization. We leave the size of each load (in bytes) up to the implementation, and use single-byte loads in our evaluation. While implementing this cache diversification technique, we identified two ways of computing the address accessed by each load instruction: (i) static address and (ii) dynamic address computation.

In the first technique, static address computation, the compiler randomly picks an address (inside the range), then hard-codes it inside the program so the load is the same for every execution:

```
offset = 0x123 // Random constant < region_size
addr = region_base + offset
tmp = Memory[addr] // Volatile load
```

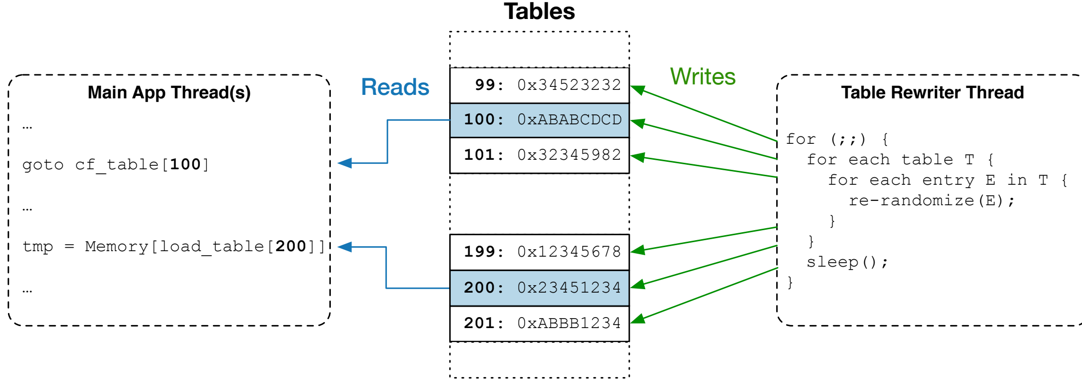

Fig. 6: Memory layout of runtime address tables, along with pseudocode of the randomization algorithm. The randomization algorithm runs periodically in an infinite loop for the entire duration of the program.

The second technique, dynamic address computation, loads addresses chosen dynamically while the program is running. We extend the same cached random tables used for control-flow diversity described below, and constantly rerandomize this table to contain valid addresses. This results in the following code for inserted load `i`:

```
addr = Memory[random_table[i]]
tmp = Memory[addr] // Volatile load
```

Static address computation requires the region size to be defined at compile time and a global variable `region_base` to be initialized at run time. The background thread for dynamic address computation randomly picks addresses for each table slot using global variables `region_base` and `region_size` that are initialized at run time.

B. Table Randomization Optimization

One of our main design goals was to make the randomized trampolines and memory loads be fast enough for practical usage. A naive initial implementation that called a random number generator for every control-flow transfer or memory operation proved to have unacceptably large overhead, even when buffering randomness. We instead chose to store branch targets and memory load addresses in tables and periodically re-randomize this table asynchronously in a background thread. At program startup we create a background thread that repeatedly iterates over all tables and randomizes each entry. Trampolines are then just a single indirect branch through a control-flow cache table, while the memory loads require an extra load from the table. Figure 6 shows the memory layout of the tables and the pseudocode of the table randomization algorithm which runs in the background thread.

Our dynamic control-flow transfer implementation could be further optimized to use inline caching and rewrite static branch instructions rather than an external table in data memory. However, branch targets are rerandomized frequently, so changing code page permissions from executable to writable and back may trump the performance improvement from inline caching. Alternatively, code pages could be left writable and executable, although this increases the risk of a code injection attack and may still be slow if instruction cache flushes are required.

IV. Evaluation

To analyze the security and performance characteristics of our techniques in a real-world setting, we evaluated dynamic control-flow diversity as a defense for the two side-channel attacks proposed by Tromer et al. [13] and discussed in Section II-B. We implemented these attacks targeting the AES-128 encryption routine in libgcrypt 1.6.1, which is the current version of the cryptographic library underlying GnuPG. Since our implementation does not currently support diversification of inline assembly, we disabled the assembly implementation of AES to force libgcrypt to use its standard C implementation. It is worth noting that this is an implementation limitation, and an industrial-strength implementation of our transformations could easily support rewriting of inline assembly as well.

To simplify our attack implementation, we made a slight change to the libgcrypt source code. We added an annotation to each of the targeted tables to force the compiler to align table entries such that no entries crossed a 64 byte cache line boundary. While both attacks could work around this alignment issue with further engineering effort, this change allowed us to more accurately measure the results of our protections.

We performed all security evaluations on an Intel Core 2 Quad Q9300 running Ubuntu 12.04 with Linux kernel 3.5.0. We targeted our attacks at the L2 cache of the processor; the Q9300 contains a 6MB L2 cache split into two halves, with each 3MB half being shared by two of the cores. The cache is 12-way set associative with 64-byte lines and 4096 sets. To minimize system interference, we stopped all unnecessary system daemons and pinned the attack to two cores, where the second core accommodated the background rewriting thread. In addition, to create the most advantageous situation possible for an attacker, our example attacks call the libgcrypt encryption function as a black box in the same process, rather than spawning or communicating with a separate process. Attacks in a more realistic setting would require even more observations to reliably extract the key, and our transformations would create additional uncertainty when coupled with the extra intra-process system noise.

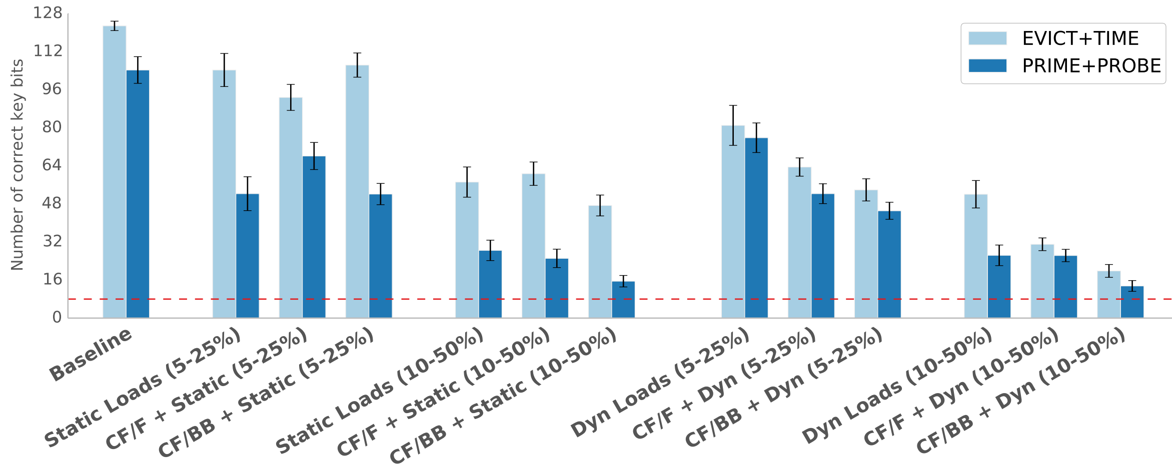

Fig. 7: Average accuracy of AES side-channel attacks with our defense. The dashed line shows the expected number of correct bits for randomly chosen keys (8 bits). Error bars represent two standard errors from the mean.

Modern processors implement a cache prefetching algorithm that assumes spatial locality of cache accesses and speculatively loads additional cache lines that the prefetching unit expects might be accessed soon. Prefetching improves performance, especially for algorithms that access long linear regions of memory. However, prefetching negatively impacts cache-based side-channel attacks by introducing the difficulty of determining which lines were loaded by the encryption algorithm and which by the prefetcher. For this reason, we disabled the prefetcher completely on our test machine by setting several configuration bits in machine status registers. While this slightly reduces the overall system performance, it makes attacks much more consistent.

We implemented all transformations and insertion of dynamic control-flow diversity as passes in version 3.3 of the Clang/LLVM compiler framework [20]. These new passes operate at the LLVM intermediate representation (IR) level, and are thus platform-independent.

A. Security Evaluation

After testing our example attacks, we empirically found that 5 million iterations of the EVICT+TIME attack and 75 thousand iterations of the PRIME+PROBE attack were sufficient to derive 96% and 82% of the random key bits on average, respectively. Although our attack implementation does not derive the full key in all cases due to random system noise and complex processor variations, this is an implementation concern, and an attacker would likely tune these attacks for increased accuracy.

To ensure that our baseline was accurate, we averaged 50 runs without any diversification, using a new random secret key for each iteration. Since our transformations rely on random choices during compilation, we tested each instance with ten different random seeds and each seed with five random keys (resulting in 50 runs total for each configuration) and report the average accuracy over all seeds and keys for each configuration.

To ensure that functions or basic blocks relevant to the AES encryption implementation were replicated, we manually inspected the libgcrypt implementation and selected nine functions that the program executes for every AES encryption. To collect comparable data for each experiment, we configured our compiler to select all nine functions (or all basic blocks in the selected functions) for replication.

We fixed the number of generated replicas for each program fragment to ten in all cases. We found that further increasing this parameter had little effect on the attack success with the number of iterations we tested. However, increasing the replica count also had no measurable effect on runtime performance, and only a moderate effect on file size. Therefore adding additional variants may be a viable option to combat increasing attacker capabilities.

Security results for both the EVICT+TIME and PRIME+PROBE attacks are found in Figure 7. We label the all static cache load variants with Static and the dynamic variants with Dyn. Control-flow diversity with function and basic-block replicas is labeled respectively with CF/F and CF/BB. We report key recovery in number of bits for clarity, however, it is important to note that both attacks derive the key in nibble-sized increments.

Static Loads: Static cache noise at a 5–25% insertion rate had little effect on the EVICT+TIME attack, resulting in 104–108 of 128 key bits recovered. Adding dynamic control-flow diversity to static noise also had little effect, since there is little timing variance between replicas when using static loads. Increasing this percentage to 10–50% had a more pronounced effect. More loads naturally imply that execution will be slower and thus more sensitive to cache collisions.

Dynamic control-flow diversity did have a significant effect on the PRIME+PROBE attack when combined with static cache loads. Function-level dynamic control-flow diversity reduced the correctly key recovered key bits from 52 with static loads to 41, and basic-block level replication furthered reduced this to 31 bits. With 10–50% cache noise insertion, we saw further reduction to 16 key bits correctly recovered using basic-block dynamic control-flow diversity.

Dynamic Loads: Dynamic loads had a larger effect on the EVICT+TIME attack. Dynamic cache noise alone at a

5–25% rate reduced the average correctly recovered key bits to 81. Adding dynamic control-flow diversity on top of this further reduced the recovered key bits to 64 and 54 for function-level and basic block-level diversity respectively. At the 10–50% insertion rate we observed similar trends, with CF/BB and dynamic loads reducing the EVICT+TIME key recovery to 20 bits. Dynamic cache loads naturally have a higher performance variation, since they require an extra indirect load to implement runtime dynamic randomness. This results in a more pronounced impact on the EVICT+TIME attack.

We observed similar trends for the PRIME+PROBE attack. While dynamic loads have some effect on the attack by themselves, they are most effective when combined with function or basic-block dynamic control-flow diversity. In the best case (CF/BB + Dyn) we observed an average correct key recovery of only 14 bits. This result is near the theoretical limit of 8 bits where an attacker gains no information from the side channel. Recovering 8 bits of the key is equivalent to an adversary randomly guessing the key by nibbles without side-channel information, since such an adversary has a 1 in 16 chance to guess each nibble correctly and each key nibble is independent for uniform random keys. This expected number of correctly guessed key bits with no knowledge is a lower bound on the accuracy of any side-channel attack, and we show this bound as a dashed line in Figure 7.

Increasing samples: To investigate whether the attacks could feasibly overcome our defense by gathering more side-channel observations, we increased the iteration count for both attacks. We found that while the attack accuracy increased marginally with 4x and 8x the number of original attack measurements, a realistic attack is still infeasible. With the CF/BB + Static (10–50%) setting, 4x iterations resulted in average correctness of 70 bits for the EVICT+TIME attack and 34 bits for the PRIME+PROBE attack. 8x iterations resulted in 42 correct key bits on average for the PRIME+PROBE attack. These results indicate that dynamic control-flow diversity is still effective in the presence of better resourced attackers, although it may require a different diversifying transformation to be more effective against the EVICT+TIME attack.

Collecting eight times more samples than in our baseline attack required about five minutes of attack time, resulted in a 1.5GiB data file, and analysis took about an hour on a high end, quad-core c3.xlarge Amazon EC2 instance. In a more realistic situation collecting many more samples than this is likely prohibitive. It is important to remember that our attack is simply encrypting a single block, with no inter-process communication or application overhead. Our tests represent a best-case scenario for an attacker. A realistic attack would target a service which is doing more work than our test attacks, and thus data collection would be far slower and noisier in practice.

B. Performance Evaluation

Most existing defenses against cache side-channel attacks, e.g., reloading sensitive tables into cache after every context switch or rewriting encryption algorithms to not use cached tables at all, introduce moderate overheads. Our transformations also marginally increase the cost of AES encryption. However we believe this overhead to be quite reasonable for an automated and general side-channel defense. To properly quantify this impact, we studied an AES micro-benchmark, a full-fledged service — Apache serving files over HTTPS using AES — and the SPEC CPU2006 benchmark suite.

Transformation	File Size (KiB)	Increase Factor
Baseline	657	1.00
Static Loads (5-25%)	657	1.00
CF/F + Static (5-25%)	702	1.07
CF/BB + Static (5-25%)	716	1.09
Dyn Loads (5-25%)	658	1.00
CF/F + Dyn Loads (5-25%)	755	1.15
CF/BB + Dyn Loads (5-25%)	727	1.11
Static Loads (10-50%)	657	1.00
CF/F + Static (10-50%)	766	1.17
CF/F + Static (10-50%)	941	1.43
CF/BB + Static (10-50%)	737	1.12
CF/BB + Static (25@10-50%)	837	1.27
Dyn Loads (10-50%)	660	1.00
CF/BB + Dyn Loads (10-50%)	759	1.15
CF/F + Dyn Loads (10-50%)	784	1.19

TABLE I: File size increase for libgcrypt, relative to a non-diversified baseline.

From this performance analysis, in conjunction with attack success, we found that the optimal trade-off between security and performance is the CF/F + Static Loads setting. The CF/BB + Static Loads setting was slightly more effective, with only a small marginal decrease in performance, and is thus also an ideal candidate setting. Using dynamic loads, while slightly more effective, has a significantly larger performance impact for comparably little marginal security benefit.

AES Micro-benchmark: We first measured the increase in time introduced by each transformation with an AES micro-benchmark. We generated ten random different versions of libgcrypt for each set of parameters, ran each version of the AES encryption function five million times on random plaintexts for each of ten different random keys and measured the number of cycles for each encryption. The first column of each group in Figure 8 shows the slowdown for the libgcrypt micro-benchmark.

We found that using function or basic-block level dynamic control-flow diversity along with static cache noise results in a performance slowdown of 1.76x–2.02x compared to the baseline AES encryption when using 10–50% cache noise insertion. Dynamic cache noise at a 5–25% rate results in similar performance, but 10–50% insertion of dynamic loads has significantly more impact on performance (2.39–2.87x slowdown).

In addition to measuring encryption time, we investigated the impact of our transformations on the size of the encryption library. While desktop disk space is currently plentiful, this is not the case for embedded or mobile systems. Many programs are also distributed over the Internet through communication links that have either bandwidth or data limits. Table I shows the impact of our transformations on the size of the libgcrypt shared object.

Application Benchmark: In the previous section we measured the performance impact on AES encryption alone, encrypting a single block. However, to get a more realistic picture of the performance impact of our techniques, we also

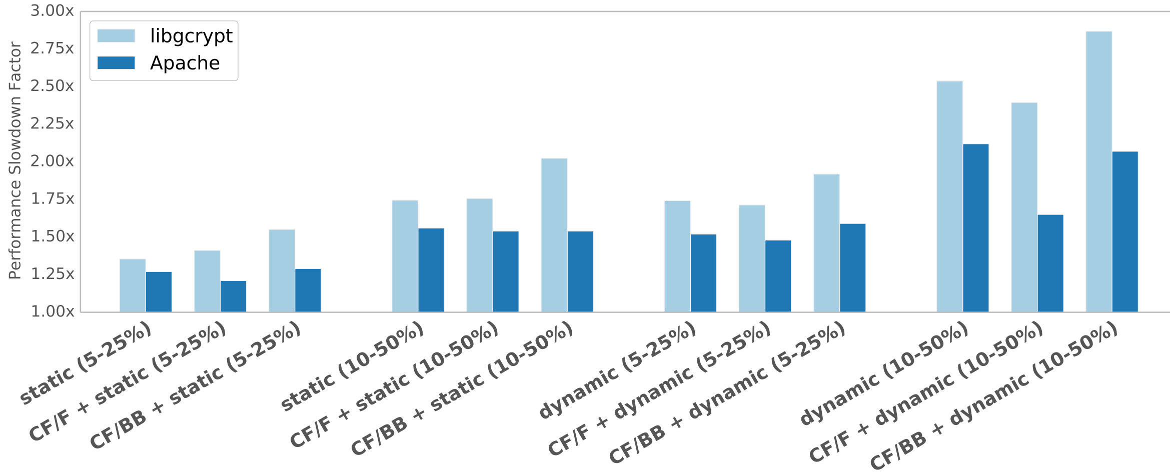

Fig. 8: Performance slowdown factor. libgcrypt: AES micro-benchmark encryption performance slowdown, relative to a non-diversified baseline. Apache: slowdown when serving a 4MB file over HTTPS with AES block cipher, relative to a non-diversified baseline.

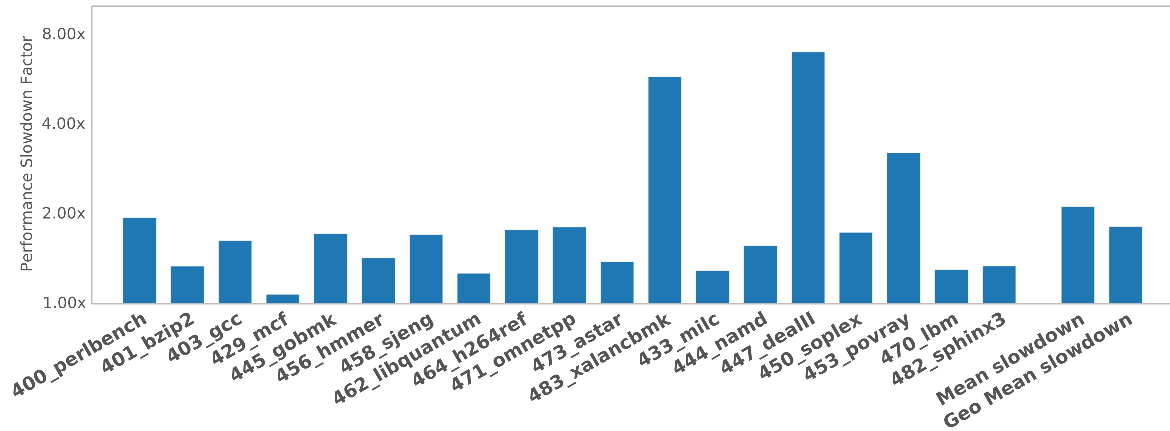

Fig. 9: Performance slowdown factor for SPEC CPU2006 with function-level dynamic control-flow diversity on 25% of functions and 10–50% static cache noise inserted in all functions. Y-axis is on a log scale.

evaluated the performance overhead of dynamic control-flow diversity and our transformations on Apache 2.4.10 serving AES encrypted data. We used the standard apachebench (ab) tool to evaluate performance, connecting over https to an Apache instance using a diversified version of the OpenSSL 1.0.1 library[2].

As seen in the second column of each group in Figure 8, the overall slowdown of our techniques varies from 1.25x for static cache noise to 2.1x for dynamic. The static noise CF/F and CF/BB settings in fact have identical overheads in this test, and we therefore recommend the CF/BB setting for practical applications which consist of more than just block cipher encryption. The overall performance impact is naturally lower than the simple micro-benchmark, since Apache does other processing in addition to encryption. However, this workload is more representative of a real-world application of cryptography and AES in particular.

SPEC CPU2006: To illustrate the effects of our techniques on CPU intensive workloads, we tested with the C and C++ portions of the SPEC CPU2006 benchmark suite. We selected one parameter setting: function-level dynamic control-flow diversity with static noise. However, since SPEC does not have any particular targets for cache side-channel attacks, we applied dynamic control-flow diversity universally over all functions with a 25% probability. We also applied static cache noise over all functions with a probability to insert noise for each instruction chosen randomly for each basic block from the range 10–50%. These parameters represent a worst-case for the CF/F + Dyn setting. To account for random choices, we built and ran SPEC with four different random seeds

As we show in Figure 9, our transformations introduce a 1.82x geometric mean overhead across all benchmarks. The xalancbmk and dealII benchmarks stand out in this test. These

[2] While we have not tested the effectiveness of the side-channel attack on this library, we believe it would take minimal effort to port the attack to OpenSSL or other table-based AES implementations.

particular benchmarks are large, complex C++ programs with many function calls. Since we applied function dynamic control-flow diversity across the entire program in this case, we naturally incur a higher overhead when the program calls many small functions. In practice users of dynamic control-flow diversity should target transformations in only the sections of code which might be vulnerable to a side-channel attack, instead.

V. Discussion

A. Parameter Settings

In our experiments we determined that a 5–25% insertion percentage range for cache noise instructions is too narrow. Dynamic control-flow diversity works best when replicas have very different runtime behavior, since it relies on switching between replicas with varying side-channel effects. In addition, libgcrypt is mostly straight line code and thus has a relatively low number of functions and basic blocks used for AES encryption. We expect that more complex cryptographic algorithms such as RSA will have more control flow, and thus more opportunity to insert dynamic control-flow diversity and switch between variants.

Cache noise, especially the dynamic variant, has an impact on execution time and thus the EVICT+TIME attack. However, this transformation is designed specifically to disrupt the PRIME+PROBE attack by polluting the cache and masking real AES table cache accesses. A transformation targeted at varying the running time of each replica would be more suited to disrupting this attack. We could adapt proposed hardware junk code insertion techniques [21], [22] to work with dynamic control-flow diversity by inserting differing code with varying runtimes into each replica.

In the best case, CF/BB + Dyn (10–50%), our EVICT+TIME attack can derive only 4.96 key nibbles, or about 20 key bits. Even with a more performance conscious alternative, CF/BB + Static (10–50%), we still prevent the attacker from finding 80 of 128 key bits. In the PRIME+PROBE attack our experiments show an average of 3.32 correctly recovered key nibbles, or 13.28 key bits, for the CF/BB + Static (10–50%) setting. The remaining approximately unknown key bits are too much to brute-force search, since this would require checking 2^n key guesses, where n is the number of unknown key bits. With this low correctness an attacker is unlikely to even be able to determine which key nibbles are correct, and thus would gain no useful information from the attack. Thus, we conclude that our techniques effectively mitigate the PRIME+PROBE attack, given a realistic attack scenario.

We chose example parameters of ten replicas for each program unit along with 5–25 and 10–50 percent probability of inserting cache noise operations at each instruction as a starting point after initial experimentation. These parameters are representative of a narrow and wider range of insertion. However, these parameters may not represent an ideal trade-off between security and performance. In fact, these parameter settings are not mutually exclusive, e.g., some functions may be diversified with static noise while others get dynamic noise. Some combination of function and basic block replicas may also be useful for some applications. For future work, we propose to develop heuristics for automatic parameter selection through application and attack profiling.

B. Disabled Cache

Disabling caching of critical memory is an often suggested naive approach to preventing cache side-channel attacks [13]. This approach is attractive since existing commodity processors support selectively disabling page caching, but unfortunately it is prohibitively slow. To verify that this mitigation is impractical, we carefully measured the performance of the AES routine in libgcrypt with caching disabled for the AES lookup tables. This required writing a custom Linux kernel module to map and mark a page of memory as uncacheable using the Page Attribute Table (PAT) available on x86 CPUs. The user mode application, in this case libgcrypt, can then map this page into its address space and store the lookup table into it. This interface, while technically possible, is complex and not available in the standard Linux kernel.

We modified libgcrypt to utilize this approach and tested the same AES micro-benchmark described above. We found that disabling caching on only the single AES lookup page caused the encryption routine to be 75 times slower than normal. Therefore disabling caching, even for a single page, is impractical on modern hardware. We discuss other hardware based cache protections in Section VI, however, these approaches are not available in commodity processors.

C. Implementation Limitations

For our initial investigation of applying control-flow diversity to side channels, we manually inspected the libgcrypt AES implementation to select nine functions relevant to the encryption algorithm. This simple step required no modification to the original sources, and could be easily automated by supplying only an encryption entry point. We forced our system to replicate these functions and their basic blocks to demonstrate the effectiveness of our techniques in a controlled environment, without the additional complication of having the system automatically select program units for diversification at random. However, this small manual effort was done to arrive at a controlled experiment and is not required to use control-flow diversity. By randomly selecting program units for replication with some configurable probability, our system can probabilistically protect an entire application from side-channel attacks with no manual effort.

Instead of random or manual program unit selection, we believe that side-channel analysis tools such as CacheAudit [23] can guide the selection of the critical program fragments and parameters for diversification. This should eliminate all manual effort while preserving a high level of security.

D. Related Attacks

Diversifying transformations, such as inserting cache noise instructions, can also be used to perform fine grained code layout randomization. This provides probabilistic protection against return-oriented programming and its variants which makes it realistic to expect that our defense technique can simultaneously defend against two or more fundamentally different classes of attacks. We will pursue this research direction in follow up work as well.

VI. Related Work

This paper unites two previously unrelated strands of research: side channels and artificial software diversity. We discuss the related work in each of these areas separately.

A. Side Channels

After Kocher described an initial timing side-channel attack on public-key cryptosystems [24], researchers have proposed a multitude of side-channel attacks against cryptographic algorithms. While researchers have proposed many different side-channel vectors ranging from power analysis [25] to acoustic analysis [26], we focus on applying our techniques against timing and cache-based attacks not requiring physical access. Cache-based attacks were first theoretically described by Page [27] in 2002. In 2003, Tsunoo et al. [28] demonstrated cache-based attacks against DES in practice. Bernstein [29] then presented a simple timing attack on AES, along with potential causes of this timing variability, including variable cache behavior and latency. Shortly after, Osvik, Shamir, and Tromer [30], [13] presented their attacks on AES, including the two example attacks used in this paper. In addition to the two synchronous attacks we evaluated our techniques against, Osvik et al. also described an asynchronous attack relying only on passively observing encryptions of plaintexts from a known but non-uniform distribution.

Recently, Hund et al. [31] used a cache-based timing side-channel attack to de-randomize kernel space ASLR in order to accurately perform code-reuse attacks in the kernel address space. Since we build our system on techniques proven to be effective against code-reuse attacks, our dynamic control-flow diversity with NOP insertion is a perfect fit to defend in depth against this attack.

As side-channel attacks have matured, researchers have proposed numerous defenses using both hardware and software. We will now briefly describe a few of the relevant defenses.

Hardware Defenses: Several different methods of preventing side channels at the hardware level have been proposed, with varying degrees of practicality. In the context of differential power analysis attacks, Irwin et al [21] proposed a new stage in the processor execution pipeline which randomly mutates the instruction stream with the assistance of a compiler-generated register liveness map. Among other peephole transformations, this mutation unit adds instructions that do not affect the correct functioning of the program, which are a super-set of our compiler-based NOP insertion transformation. Since our transformations in software are similar to the techniques Irwin et al. applied to differential power analysis, we expect that our technique will apply directly to power analysis attacks as well. Finally, Irwin et al. proposed a new probabilistic branch instruction, *maybe*, that would allow us to efficiently randomize control flow without the use of a random buffer. Ambrose et al. [22] also proposed inserting random instructions but with the added requirement that inserted instructions modify processor state, e.g., registers, so the new instructions are indistinguishable from legitimate program code.

To specifically target cache-based attacks, Page [32] proposed partitioning the cache into disjoint configurable sets so that a sensitive program cannot share cache resources with an attacker. However this would require a radical change to current cache designs. Bernstein [29] suggested the addition of a new CPU instruction to load an entire table into L1 cache and perform a lookup. This approach provides consistent cache access behavior regardless of input, and as such would eliminate cache side channels through table lookups. Wang and Lee [33] also proposed two new hardware cache designs to mitigate cache side channels: PLcache and RPcache. PLcache has the new capability of locking a sensitive cache partition into cache, while RPcache randomizes the mapping from memory locations to cache sets. While these techniques are powerful mitigations against cache side-channel attacks, they all require additional hardware features which major processor vendors are unlikely to implement. In contrast, our techniques require no special hardware support and can be used immediately.

Intel has recently implemented a new hardware instruction to perform encryption and decryption for AES [34]. Since this instruction is data independent, using it instead of a software routine should protect against side-channel attacks on AES. However, this hardware only implements AES, and thus we still need defensive measures to protect other cryptographic algorithms.

Software Defenses: The ideal defense against side-channel attacks is to modify the sensitive program so that it has no input-dependent side-effects, however this is an extremely labor-intensive solution and is often infeasible. Developers generally take this approach to removing timing side channels by creating algorithms that run in constant-time regardless of inputs. Bernstein [29] strongly recommends this approach, while cautioning that software which the programmer expected to run in constant time may not do so due to hardware complexity.

Page [35] suggested manually adding noise to encryption to make cache side-channel attacks more difficult in a manner conceptually similar to our automatic randomizing transformations. For instance, Page manually inserted garbage instructions and random loads into the encryption routine to combat timing and trace based attacks respectively. Page's work is a form of obfuscation rather than diversification since all users run the same binaries with the same runtime control-flow. Our combination of control flow randomization and garbage code insertion *simultaneously* defends against code reuse attacks and side channels whereas garbage code in itself does not protect against side channels and Page's transformations do not protect against code reuse.

Brickell et al. [36] proposed the use of compressed and randomized tables for AES that would alleviate cache-based attacks. However, this implementation process requires manually rewriting the AES implementation and is specific to the operation of AES.

Cleemput et al. [37] proposed defenses that do not require manual program modification. In particular, they described the use of compiler transformations to reduce timing variability. Our approach, while also compiler-based, seeks to mask variability rather than remove it entirely, since opportunities to automatically eliminate variable-time routines are limited.

In their recent paper addressing side-channel attacks in the context of virtualized cloud computing, Zhang and Reiter [38] proposed periodically scrubbing shared caches used by sensitive

processes. This scheme potentially breaks cache snooping by a time-shared process on the same core, but will not necessarily combat cache attacks in a Simultaneous Multithreading (SMT) context or continuous power analysis attacks. Since our random decision points are more fine grained than the scrubbing interval, our techniques have greater potential against these fine-grained attacks, although this would require more investigation. In addition, control-flow diversity does not depend on any resources outside the program and is thus applicable in situations without hypervisors, such as embedded software.

Finally, Tromer et al. [13] mention adding noise to memory accesses with spurious accesses to decrease the signal available to the attacker. Effectively, our technique accomplishes this goal in a general way that could be extended to other side channels, and we provide a concrete evaluation showing its effectiveness in practice. Since adding replicas exponentially increases the number of possible execution traces, we can ratchet our defense up sufficiently so that an attacker cannot feasibly collect and analyze enough samples.

B. *Artificial Software Diversity*

The literature on artificial software diversity is extensive; we limit ourselves to the work most closely related to ours. Larsen et al. provides a comprehensive systematization of approaches to artificial software diversity [39]. Cohen initially pioneered software diversity as a protection against reverse engineering [1] and was first to suggest garbage code insertion and transformations that obscure the actual control flow. Collberg et al. [40] extended these ideas into a broader set of obfuscating transformations against reverse engineering attacks and introduced the notion of opaque predicates [41]. While opaque predicates usually refer to predicates that have a known outcome at obfuscation time but are expensive to decide afterward via static analysis, Collberg et al. also mention "variable" opaque predicates that flip-flop between true and false at runtime. These ideas were evaluated in depth by Anckaert et al. [12] as a defense against reverse engineering, by Collberg et al. [8] in context of client-side tampering of networked systems, and by Coppens et al. [7] to prevent reverse engineering of patches. Our work differs in its use of control flow randomization: we use it to switch among implementation variants (replicas) with fine-granularity—not as a randomizing transformation in itself. Furthermore, we aim to thwart side-channel attacks rather than reverse engineering.

Several diversified defenses against code reuse attacks have dynamic aspects. Giuffrida et al. [6] presented a compiler-based approach that periodically rerandomizes services in a microkernel OS while it is running. Live rerandomization works by periodically transferring the application state from one process to another such that the old and new processes run diversified variants of the same input program. While this provides excellent protection against code reuse attacks, the rerandomization overhead prevents the fine granularity our approach efficiently supports.

Hiser et al. [4] performed fine-grained code layout randomization using a process virtual machine. The approach uses a code cache that leads to predictable program traces and might constitute a side channel in itself. Homescu et al. [11] diversifies just-in-time compiled code and similarly caches translated code to improve performance. Shioji et al. [10] introduced "code shredding" that embeds random checksums in pointers to thwart control-flow hijacking. To improve performance and add randomness, checksums are not masked out before the pointer values are used in control flow transfers. Rather, the entire code section is replicated in process memory to make the targets of checksummed pointers valid. Our use of code replication is more flexible because our granularity can vary at the function or basic block level and has a lower memory overhead as a result. Our performance overhead is also much lower since our compiler-based approach avoids the overheads associated with binary rewriting; Shioji report overheads ranging from 3x to 26x on Bzip2 1.0.5.

Novark and Berger secure the heap against memory management errors via a randomizing memory allocator [9]. Allocations are placed randomly in memory and stay in place until their deallocation. Freed pages are overwritten with random data. While this can interfere with side-channel attacks, attackers can sample the victim process arbitrarily many times between memory allocator activations.

Summing up, our work is the first to use software diversity to mitigate cache side-channel attacks. Previous diversification approaches comprise one or more randomizing code transformations. Our approach consists of a runtime randomization mechanism to dynamically vary execution characteristics *in addition to* a set of randomizing code transformations.

VII. Conclusion and Outlook

We provide the first evaluation of software diversity as a side-channel mitigation. To that end, we developed dynamic control-flow diversity which performs fine-grained program trace randomization. Our technique does not require source code modification or specialized hardware so it can be automatically applied to existing software. We have implemented a prototype diversifier atop LLVM and rigorously evaluated the performance of our techniques using modern, realistic cache side-channel attacks in a setting that favors attackers. Our experimental evaluation shows that our technique mitigates cryptographic side channels with high efficacy and moderate overhead of 1.5–2x in practice, making it viable for deployment.

Beyond the cryptographic side-channel problem addressed in this paper, we expect that control-flow diversity is simultaneously effective against other implementation-dependent attacks, including code reuse and reverse engineering. We plan to explore this in future work.

Acknowledgments

We thank the anonymous reviewers for their insightful comments and suggestions. We are also grateful for helpful feedback from Mathias Payer and Mark Murphy.

This material is based upon work partially supported by the Defense Advanced Research Projects Agency (DARPA) under contracts D11PC20024 and N660001-1-2-4014, and generous gifts from Mozilla and Oracle. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the Defense Advanced Research Projects Agency (DARPA), its Contracting Agents, or any other agency of the U.S. Government.

References

[1] F. Cohen, "Operating system protection through program evolution," *Computers and Security*, vol. 12, no. 6, pp. 565–584, Oct. 1993.

[2] S. Forrest, A. Somayaji, and D. Ackley, "Building diverse computer systems," in *Proceedings of the 6th Workshop on Hot Topics in Operating Systems (HOTOS '97)*, 1997, pp. 67–72.

[3] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, 2012, pp. 601–615.

[4] J. Hiser, A. Nguyen-Tuong, M. Co, M. Hall, and J. W. Davidson, "ILR: Where'd my gadgets go?" in *Proceedings of the 33rd IEEE Symposium on Security and Privacy (S&P '12)*, 2012, pp. 571–585.

[5] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*. ACM, 2012, pp. 157–168.

[6] C. Giuffrida, A. Kuijsten, and A. S. Tanenbaum, "Enhanced operating system security through efficient and fine-grained address space randomization," in *Proceedings of the 21st USENIX Security Symposium*, 2012, pp. 475–490.

[7] B. Coppens, B. De Sutter, and J. Maebe, "Feedback-driven binary code diversification," *ACM Transactions on Architecture and Code Optimization*, vol. 9, no. 4, pp. 24:1–24:26, Jan. 2013.

[8] C. S. Collberg, S. Martin, J. Myers, and J. Nagra, "Distributed application tamper detection via continuous software updates," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, 2012, pp. 319–328.

[9] G. Novark and E. D. Berger, "Dieharder: securing the heap," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS '10)*. ACM, 2010, pp. 573–584.

[10] E. Shioji, Y. Kawakoya, M. Iwamura, and T. Hariu, "Code shredding: byte-granular randomization of program layout for detecting code-reuse attacks," in *Proceedings of the 28th Annual Computer Security Applications Conference (ACSAC '12)*, 2012, pp. 309–318.

[11] A. Homescu, S. Brunthaler, P. Larsen, and M. Franz, "librando: Transparent code randomization for just-in-time compilers," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*. ACM, 2013, pp. 993–1004.

[12] B. Anckaert, M. Jakubowski, R. Venkatesan, and K. D. Bosschere, "Run-time randomization to mitigate tampering," in *Proceedings of the 2nd International Workshop on Security (IWSEC '07)*, 2007, pp. 153–168.

[13] E. Tromer, D. A. Osvik, and A. Shamir, "Efficient cache attacks on AES, and countermeasures," *Journal of Cryptology*, vol. 23, no. 1, pp. 37–71, Jan. 2010.

[14] Y. Zhang, A. Juels, M. K. Reiter, and T. Ristenpart, "Cross-VM side channels and their use to extract private keys," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*. ACM, 2012, pp. 305–316.

[15] A. Homescu, S. Neisius, P. Larsen, S. Brunthaler, and M. Franz, "Profile-guided automatic software diversity," in *Proceedings of the 11th IEEE/ACM International Symposium on Code Generation and Optimization (CGO '13)*, 2013, pp. 1–11.

[16] D. Gullasch, E. Bangerter, and S. Krenn, "Cache games - bringing access-based cache attacks on AES to practice," in *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P '11)*, 2011, pp. 490–505.

[17] Y. Yarom and K. Falkner, "Flush+reload: a high resolution, low noise, L3 cache side-channel attack," Cryptology ePrint Archive, Report 2013/448, 2013.

[18] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, "Hey, you, get off of my cloud: Exploring information leakage in third-party compute clouds," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS '09)*. ACM, 2009, pp. 199–212.

[19] M. Saito and M. Matsumoto, "SIMD-Oriented fast mersenne twister: a 128-bit pseudorandom number generator," in *Monte Carlo and Quasi-Monte Carlo Methods 2006*, A. Keller, S. Heinrich, and H. Niederreiter, Eds. Springer Berlin Heidelberg, Jan. 2008, pp. 607–622.

[20] C. Lattner and V. Adve, "LLVM: A compilation framework for lifelong program analysis & transformation," in *Proceedings of the 2nd IEEE/ACM International Symposium on Code Generation and Optimization (CGO '04)*, 2004, pp. 75–87.

[21] J. Irwin, D. Page, and N. Smart, "Instruction stream mutation for non-deterministic processors," in *Proceedings of the 13th IEEE International Conference on Application-Specific Systems, Architectures and Processors (ASAP '02)*, 2002, pp. 286–295.

[22] J. A. Ambrose, R. G. Ragel, and S. Parameswaran, "RIJID: random code injection to mask power analysis based side channel attacks," in *Proceedings of the 44th Design Automation Conference (DAC '07)*. ACM, 2007, pp. 489–492.

[23] G. Doychev, D. Feld, B. Köpf, L. Mauborgne, and J. Reineke, "Cacheaudit: A tool for the static analysis of cache side channels," in *Proceedings of the 22nd USENIX Security Symposium*, 2013, pp. 431–446.

[24] P. C. Kocher, "Timing attacks on implementations of diffie-hellman, RSA, DSS, and other systems," in *Advances in Cryptology (CRYPTO '96)*, ser. Lecture Notes in Computer Science, N. Koblitz, Ed. Springer Berlin Heidelberg, Jan. 1996, no. 1109, pp. 104–113.

[25] P. Kocher, J. Jaffe, and B. Jun, "Differential power analysis," in *Advances in Cryptology (CRYPTO '99)*, ser. Lecture Notes in Computer Science, M. Wiener, Ed. Springer Berlin Heidelberg, Jan. 1999, no. 1666, pp. 388–397.

[26] D. Genkin, A. Shamir, and E. Tromer, "RSA key extraction via low-bandwidth acoustic cryptanalysis," in *Advances in Cryptology (CRYPTO '14)*, ser. Lecture Notes in Computer Science, J. A. Garay and R. Gennaro, Eds. Springer Berlin Heidelberg, Jan. 2014.

[27] D. Page, "Theoretical use of cache memory as a cryptanalytic side-channel," Cryptology ePrint Archive, Report 2002/169, 2002.

[28] Y. Tsunoo, T. Saito, T. Suzaki, M. Shigeri, and H. Miyauchi, "Cryptanalysis of DES implemented on computers with cache," in *Cryptographic Hardware and Embedded Systems (CHES '03)*, ser. Lecture Notes in Computer Science, C. D. Walter, Ç. K. Koç, and C. Paar, Eds. Springer Berlin Heidelberg, Jan. 2003, no. 2779, pp. 62–76.

[29] D. J. Bernstein, "Cache-timing attacks on AES," Preprint, 2005.

[30] D. A. Osvik, A. Shamir, and E. Tromer, "Cache attacks and countermeasures: The case of AES," in *Topics in Cryptology (CT-RSA '06)*, ser. Lecture Notes in Computer Science, D. Pointcheval, Ed. Springer Berlin Heidelberg, Jan. 2006, no. 3860, pp. 1–20.

[31] R. Hund, C. Willems, and T. Holz, "Practical timing side channel attacks against kernel space ASLR," in *Proceedings of the 34th IEEE Symposium on Security and Privacy (S&P '13)*, 2013.

[32] D. Page, "Partitioned cache architecture as a side-channel defence mechanism," Cryptology ePrint Archive, Report 2005/280, 2005.

[33] Z. Wang and R. B. Lee, "New cache designs for thwarting software cache-based side channel attacks," in *Proceedings of the 34th International Symposium on Computer Architecture (ISCA '07)*. ACM, 2007, pp. 494–505.

[34] S. Gueron, "Intel advanced encryption standard (AES) instructions set," *Intel White Paper*, 2010.

[35] D. Page, "Defending against cache-based side-channel attacks," *Information Security Technical Report*, vol. 8, no. 1, pp. 30–44, 2003.

[36] E. Brickell, G. Graunke, M. Neve, and J.-P. Seifert, "Software mitigations to hedge AES against cache-based software side channel vulnerabilities." Cryptology ePrint Archive, Report 2006/052, 2006.

[37] J. V. Cleemput, B. Coppens, and B. De Sutter, "Compiler mitigations for time attacks on modern x86 processors," *ACM Transactions on Architecture and Code Optimization*, vol. 8, no. 4, pp. 23:1–23:20, Jan. 2012.

[38] Y. Zhang and M. K. Reiter, "Düppel: Retrofitting commodity operating systems to mitigate cache side channels in the cloud," in *Proceedings of the 20th ACM Conference on Computer and Communications Security (CCS '13)*. ACM, 2013, pp. 827–838.

[39] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014, pp. 276–291.

[40] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, University of Auckland, New Zealand, Tech. Rep. 148, 1997.

[41] C. S. Collberg, C. D. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM Symposium on Principles of Programming Languages (POPL '98)*, 1998, pp. 184–196.