No Forking Way: Detecting Cloning Attacks on Intel SGX Applications

Anonymous Author(s)

ABSTRACT

Forking attacks against TEEs like Intel SGX can be carried out either by rolling back the application to a previous state, or by cloning the application and by partitioning its inputs across the cloned instances. Current solutions to forking attacks require Trusted Third Parties (TTP) that are hard to find in real-world deployments. In the absence of a TTP, many TEE applications rely on monotonic counters to mitigate forking attacks based on rollbacks; however, they have no protection mechanism against forking attack based on cloning. In this paper, we analyze 72 SGX applications and show that approximately 20% of those are vulnerable to forking attacks based on cloning—including those that rely on monotonic counters.

To address this problem, we present CloneBuster, the first practical clone-detection mechanism for Intel SGX that does not rely on a TTP and, as such, can be used directly to protect existing applications. CloneBuster allows enclaves to (self-) detect whether another enclave with the same binary is running on the same platform. To do so, CloneBuster relies on a cache-based covert channel for enclaves to signal their presence to (and detect the presence of) clones on the same machine. We show that CloneBuster is robust despite a malicious OS, only incurs a marginal impact on the application performance, and adds approximately 800 LoC to the TCB. When used in conjunction with monotonic counters, CloneBuster allows applications to benefit from a comprehensive protection against forking attacks.

1 INTRODUCTION

Trusted Execution Environments (TEE), such as Intel SGX, enable user processes to run in isolation (i.e., in so-called enclaves) from other software on the same platform, including the OS. Intel SGX applications are, however, susceptible to so-called *forking attacks*, where the adversary partitions the set of clients and provides them with different views of the system. Forking attacks may be mounted either by cloning an enclave or by rolling back its state [61]. Rollback attacks exploit the fact that the sealing functionality of Intel SGX lacks freshness guarantees. This opens the door for a malicious OS to feed a victim enclave with stale state, whenever the enclave requests to unseal its state from storage—thereby "rolling back" the enclave to a previous state. Cloning attacks leverage the fact that Intel SGX does not provide means to control the number of enclaves, with the same binary, that a malicious OS can launch on the same machine.

Forking attacks against enclaves—either by rollback or by cloning—result in serious consequences in a number of applications ranging from digital payments [124] to password-based authentication [167]. For example, in a password manager application, forking attacks may allow an adversary to brute-force a password despite rate-limiting measures adopted by the application. Similarly, in a payment application, an adversary could spend the same coins in multiple payments by reverting the state of its account balance.

Problem. A comprehensive solution to thwart forking attacks requires a centralized trusted third party (TTP) [177] or a distributed one [61, 109, 131, 139]. Unfortunately, in most real-world applications, TTPs are hard to find. Moreover, some TTP-based solutions might themselves be subject to cloning attacks during the initialization process, unless the initialization involves yet another TTP (e.g., a trusted administrator [131] or a blockchain [139]). Without TTPs, most applications can mitigate forking attacks based on rollbacks by means of hardware-based monotonic counters [167]. However, an application that uses monotonic counters can still be cloned—making it still susceptible to forking attacks. To confirm this intuition, we thoroughly analyzed the security of 72 SGX-based proposals listed in [13, 24] with respect to forking attacks. Our findings show that 14 of those applications (i.e., roughly 20%) are vulnerable to forking attacks based on cloning. Among those vulnerable proposals, only 3 rely on monotonic counters to counter rollback attacks, but can still be forked by cloning. A notable (production-ready) application that is vulnerable to forking by cloning is BI-SGX [153]. The latter has been previously found vulnerable to forking attacks based on rollbacks [107]; the authors of [107] propose to fix the vulnerability using monotonic counters. We show that relying on monotonic counters is not enough to prevent forking attacks and report a forking attack based on cloning against the fixed version of BI-SGX that uses monotonic counters.

Research question. *Can we design an anti-cloning solution that is practical, efficient, and does not require a TTP?* To the best of our knowledge, no such solution exists at the moment.

Concrete solution. To address this question, we propose CloneBuster, the first practical clone detection mechanism for SGX enclaves that does not rely on any external party. CloneBuster provides enclaves with the ability to (self-) detect whether other enclaves with the same binary are running on the same platform—without relying on a TTP. More precisely, we show how to leverage cache-based covert channels as a signaling mechanism for enclaves. Intuitively, if each enclave running on a machine uses the same channel to signal its presence to (and detect the presence of) other enclaves loaded with the same binary, cloning attacks can be promptly detected. CloneBuster ensures robust detection of clones despite noise on the channel—due to other benign applications polluting the cache—and even *when the OS is malicious*. When used in conjunction with monotonic counters, CloneBuster enables enclaves to benefit from a comprehensive protection against all types of forking attacks (including rollback attacks) without relying on an external trusted party. Moreover, we show that CloneBuster could be equally used in solutions like ROTE [131] or NARRATOR [139] to avoid the use of yet another TTP when the system is being initialized. We summarize our contributions as follows:

Impact of cloning on SGX applications: We thoroughly analyze the vulnerability of 72 SGX-based applications against forking attacks (cf. Section 3). We show that 14 applications either do not account for any protection mechanism against forking or simply prevent forking attacks based on rollbacks by means of monotonic counters—these remain vulnerable to forking attacks based on cloning. Inspired by these findings, we discuss in details how to mount a forking attack based on cloning against such applications. We also describe and implement an attack against a production-ready open-source application.

CloneBuster: We introduce a practical, novel clone-detection mechanism, dubbed CloneBuster, that does not rely on a TTP (cf. Section 4). We analyze the security of CloneBuster and show that it can effectively detect clones in spite of a malicious OS (cf. Section 5). We also show how to extend CloneBuster to allow the detection of up to $n > 1$ clones and therefore cater for the cases where the application logic accommodates for up to $n - 1$ clones. Finally, we show that CloneBuster can be equally used to secure the initialization process of existing TTP-based forking solutions, such as ROTE [131] and NARRATOR [139].

Prototype implementation & evaluation We implemented a prototype of CloneBuster and evaluated it under realistic workloads (cf. Section 7). We additionally report the performance of CloneBuster when used to detect forking attacks on an open-source production-ready SGX application. Our evaluation results show that CloneBuster achieves high detection (F1 score up to 0.999), with a maximum performance penalty of 4%; the TCB increase is only 800 LoC. The code of our prototype is available at [50]

2 BACKGROUND

2.1 Intel SGX

Intel Software Guard Extensions (SGX) is an x86 instruction set extension that offers hardware-based isolation to trusted applications that run in so-called *enclaves* [99]. Enclave isolation leverages dedicated, hardware-protected memory called Enclave Page Cache (EPC). The OS is responsible for loading the enclave's software in the EPC while the processor keeps track of deployed enclaves and their memory pages in the Enclave Page Cache Map (EPCM). The latter allows the hardware to restrict access to EPC from processes running at higher privilege levels, including the OS or the hypervisor. In particular, the Memory Management Unit (MMU) uses the EPCM to abort any attempt to access the enclave memory that has not been issued by the enclave itself or that does not comply with the specified read/write/execute permissions.

Attestation allows the platform to issue publicly verifiable statements of the software configuration of an enclave. In particular, each application enclave has two identities: one called MRENCLAVE computed as the hash of the enclave binary loaded into memory; the other identity is called MRSIGNER and identifies the enclave developer. During attestation, a designated system enclave outputs a signature over both identities to certify that the application runs in an enclave on an SGX-enabled platform.

Intel SGX also allows enclaves to store encrypted data on disk. This is achieved via a "sealing" interface that uses hardware-managed cryptographic keys. Sealed data is encrypted and authenticated using keys that are dependent on the platform and on one of the enclave identities. Sealing data against the MRENCLAVE identity ensures that only enclaves loaded with the same binary on the same platform can unseal it; on the other hand, sealing data against the MRSIGNER identity ensures that all enclaves running on the same platform and issued by the same developer (hence, with the same MRSIGNER) can unseal it.

We note that Intel SGX has been deprecated in last-generation CPUs for desktops but will still be available for server-grade platforms [102], which fits the confidential computing in the cloud paradigm. Further, the size of the EPC will increase up to 64GB.

2.2 Cloning SGX Enclaves

Cloning an application (irrespective of whether it resides within an enclave) may or may not include its runtime memory. "Live" cloning consists of creating a copy of a running process, that includes also the runtime memory of the original process. In contrast, a "non-live" cloning operation creates a clone by only copying the code and the persistent state.

We note that Intel SGX limits live cloning of enclaves "by design". In particular, EPC encrypted memory and hardware-managed EPCM prevent live cloning of enclaves: in a nutshell, an encrypted memory page assigned to a given enclave, cannot be copied and assigned to another one.

With respect to non-live cloning, we note that the sealing functionality used to persist state information to disk prevents cross-platform cloning. In particular, cryptographic keys that Intel SGX uses for sealing enclave data, depend on the host where the enclave is running. Therefore, state sealed by an enclave on a given host cannot be unsealed on a different host.

Nevertheless, Intel SGX does not prevent non-live cloning of an enclave on the same platform, nor does it provide a mechanism to distinguish two such clones. In particular, the number of enclaves that can be set up on a given host and executed at the same time—regardless of the loaded binary—is only limited by system resources. Thus, little prevents an adversary, that controls the OS on a given host, to launch a number of enclaves with the same binary. In case one of those enclaves seals data to disk, all other enclaves with the same binary have access to that data—since sealing keys on a given host only depend on the enclave identity. As a result, if one enclave is attested and provisioned with a secret, all clones will have access to the same secret. Intel acknowledges that there is no mechanism to distinguish enclaves loaded with the same binary on the same platform, since they all share the same identities (i.e., MRSIGNER and MRENCLAVE).[1]

3 CLONING ATTACKS ON INTEL SGX

3.1 Motivation

Forking attacks against TEEs such as Intel SGX can be mounted either by rolling back the enclave to a previous state or by launching several instances of the victim enclave [61].

To illustrate how forking attacks based on cloning work, assume an enclave that is not susceptible of rollback attacks—e.g., an enclave that uses monotonic counters to seal its state. We can model the enclave as an automata E_{ID}, where ID refers to the identity of

[1] https://intel.ly/3uprwdh

Project	Source code available	Vulnerable to	
		Rollback	Cloning
Encrypted Databases and Key-value Stores			
Aria [186] ^{p}	No	N/A	Yes (A)
Avocado [38, 55] ^{a}	Yes	N/A	Yes (A)
Enclage [169] ^{p}	No	N/A	Yes (A)
EnclaveCache [67] ^{p}	No	Yes	Yes (B)
EnclaveDB [147] ^{p}	No	No (MC)	No (TTP)
HardIDX [86] ^{p}	No	Yes	N/A
NeXUS [7, 76] ^{p}	Yes	No (MC)	Yes (B)
ObliDB [8, 83] ^{p}	Yes	N/A	Yes (A)
PESOS [114] ^{p}	No	N/A	N/A
SeGShare [87] ^{p}	No	No (MC)	N/A
ShieldStore [22, 111] ^{ap}	Yes	No (MC)	Yes (B)
SPEICHER [1, 56] ^{a}	Yes	No (MC)	No (TTP)
STANlite [35, 154] ^{ap}	Yes	N/A	Yes (A)
StealthDB [15, 180] ^{a}	Yes	Yes	Yes (B)
Applications			
BI-SGX [16] ^{a}	Yes	No (MC)	Yes (B)
CACIC [49, 170] ^{a}	Yes	Yes	Yes (B)
DEBE [44, 187] ^{a}	Yes	N/A	N/A
HySec-Flow [39, 182] ^{a}	Yes	N/A	N/A
PrivaTube [73] ^{p}	No	N/A	Yes (C)
REX [47, 75] ^{a}	Yes	N/A	N/A
SGXDedup [37, 150] ^{a}	Yes	N/A	N/A
Signal CDS [11, 130] ^{a}	Yes	N/A	N/A
SkSES [23, 113] ^{a}	No	N/A	N/A
SMac [34, 79] ^{a}	Yes	N/A	N/A
TresorSGX [5, 151] ^{a}	Yes	N/A	N/A
Key + Password Management			
DelegaTEE [132] ^{p}	No	No (MC)	N/A
FeIDo [45, 158] ^{a}	Yes	N/A	N/A
Keys in Clouds [17, 116] ^{a}	Yes	No (MC)	N/A
SafeKeeper [21, 115] ^{a}	Yes	No (MC)	N/A
SGX-KMS [12, 64] ^{a}	Yes	Yes	Yes (B)
Private Search			
BISEN [25, 84] ^{a}	Yes	N/A	N/A
DeSearch [40, 122] ^{a}	Yes	N/A	N/A
Maiden [33, 181] ^{a}	Yes	N/A	N/A
POSUP [20, 95] ^{a}	Yes	N/A	N/A
QShield [30, 70] ^{a}	Yes	N/A	N/A
Snoopy [42, 74] ^{a}	Yes	No (MC)	N/A
X-Search [36, 138] ^{ap}	Yes	N/A	Yes (C)
Blockchains			
BITE [133] ^{p}	No	No (MC)	N/A
BLOXY [152] ^{p}	No	N/A	N/A
Ekiden [6, 71] ^{a}	Yes	No (TTP)	No (TTP)
Hybrids on Steroids [57] ^{p}	No	No (MC+TTP)	No (TTP)
MobileCoin [41, 85] ^{a}	Yes	No (TTP)	No (TTP)
Oasis [19] ^{a}	Yes	No (TTP)	No (TTP)
Obscuro [9, 174] ^{a}	Yes	N/A	N/A
Phala Network [27, 190] ^{a}	Yes	No (TTP)	No (TTP)
Private Chaincode [60, 97] ^{a}	Yes	N/A	N/A
Private Data Objects [59, 118] ^{a}	Yes	No (TTP)	No (TTP)
Proof of Luck [2, 136] ^{a}	Yes	N/A	No ($\star$)
Teechain [123, 125] ^{ap}	Yes	No (MC)	N/A
Town Crier [4, 192] ^{a}	Yes	No (TTP)	No (TTP)
Troxy [57] ^{p}	No	N/A	No (TTP)
Twilight [46, 80] ^{a}	Yes	N/A	N/A
Machine Learning			
Confidential ML [108] ^{a}	Yes	N/A	N/A
DP-GBDT [129] ^{a}	Yes	No (MC)	N/A
Plinius [29, 191] ^{a}	Yes	No (MC)	N/A
secureTF [149] ^{p}	No	N/A	N/A
Secure XGBoost [31, 121] ^{a}	Yes	N/A	N/A
Slalom [172, 173] ^{ap}	Yes	N/A	N/A
SOTER [43, 162] ^{a}	Yes	N/A	N/A
Network			
ConsenSGX [26, 155] ^{a}	No	N/A	N/A
CYCLOSA [144] ^{p}	No	N/A	N/A
ENDBOX [90] ^{ap}	No	N/A	N/A
LightBox [18, 81] ^{ap}	Yes	N/A	N/A
SENG [32, 159] ^{a}	Yes	N/A	N/A
SGX CBR [145] ^{p}	No	N/A	N/A
SGX-Tor [14, 110] ^{ap}	Yes	Yes	N/A
TEE V2V [48, 106] ^{a}	No	N/A	N/A
MACSec [112] ^{a}	Yes	No (MC)	N/A
S-NFV [163] ^{p}	No	N/A	N/A
SafeBricks [3, 146] ^{ap}	Yes	N/A	N/A
SELIS-PubSub [28, 54] ^{p}	Yes	N/A	N/A
Data Analytics			
Opaque [10, 193] ^{a}	Yes	No (TTP)	No (TTP)

Table 1: We analysed SGX applications listed in [13] (superscript p next to the citation) and listed in [24] (superscript a next to the citation). We excluded libraries, runtime frameworks, and projects without documentation. We divide the remaining ones based on their type (Machine Learning, Blockchain, Encrypted Databases, ...). For each application, we report whether the code is available, whether they are vulnerable to rollback attacks, and whether they are vulnerable to cloning attacks (highlighted in gray). In case the application is not vulnerable to a specific attack, we report the countermeasure (MC is monotonic counters, TTP is trusted third party). We use N/A in case the attack is not applicable. In case of applications vulnerable to cloning attacks, we categorize the attack type (A, B, C) and provide more details in Appendix E. Proof of Luck ($\star$) is not vulnerable to cloning because it books all MCs on the platform at startup; as a result, no clone can be started but this also means that MCs are no longer available for other applications on the same host.

the enclave (i.e., MRSIGNER and MRENCLAVE). Upon start, the enclave obtains the initial state S_0 from the OS and it is ready to process inputs. The enclave moves to the next state S_j as a function F of the current state and the current input I_j. For example, without malicious interference, an enclave fed with inputs I_1, I_2, and I_3 (in that order), moves through states $S_1 = F(S_0, I_1)$, $S_2 = F(S_1, I_2)$, and final state $S_3 = F(S_2, I_3)$. Each time the enclave moves to a new state, it seals the new state to disk so to resume from the latest state upon reboot.

To fork the application, the adversary can create two clones, say E_{ID} and E'_{ID}, and provide both of them with initial state S_0. Next, the OS feeds inputs I_1 and I_2 to E_{ID} and it feeds I_3 to E'_{ID}. Thus, enclave E_{ID} moves to state $S_1 = F(S_0, I_1)$ and final state $S_2 = F(S_1, I_2)$, whereas E'_{ID} move to state $S'_3 = F(S_0, I_3)$. The above example implies that a successful forking attack based on cloning

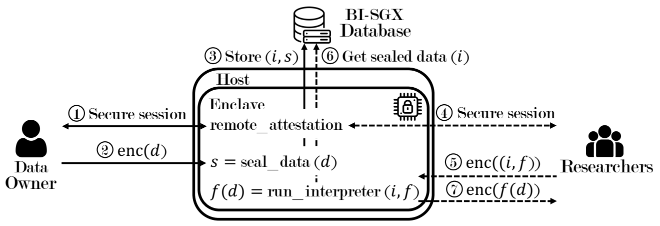


Figure 1: Overview of the BI-SGX enclave and its interactions with Data Owners and Researchers

requires running multiple instances of the victim enclave *at the same time between two state updates*. Running the two instances one at a time does not lead to a fork. To illustrate this, assume E'_{ID} is started *after* that E_{ID} has processed input I_2 and sealed state S_2. Thus, upon start E'_{ID} fetches the latest state S_2 from disk—recall that the application is not susceptible to rollbacks— obtains input I_3 and moves to state $S_3 = F(S_2, I_3)$.

Comprehensive solutions to forking attacks rely on a centralized [177] or distributed TTP [61, 109, 131, 139, 177]. For example, the authors of [61] show how to detect forking attacks if clients are mutually trusted—that is, clients themselves act as a distrusted TTP. Solutions like ROTE [131] or NARRATOR [139] prevent forking attacks by using a cohort of enclaves—distributed across different hosts—that offer forking prevention to (other) application enclaves. It is interesting to note that solutions like ROTE can be themselves victim of forking attacks by cloning when the cohort of enclaves is being initialized [139]. Once the cohort is forked, applications enclaves that use ROTE can be forked. ROTE [131] prevents forks of the cohort during initialization by means of a trusted administrator that helps initializing the cohort; NARRATOR removes the need for a centralized TTP—the administrator—by replacing it with a BFT-like blockchain, thereby using a distributed TTP.

This results in the following observation: *some TTP-based solution to forking like NARRATOR needs to use another TTP (i.e., the blockchain) to avoid being forked during its initialization process.* As such, existing solutions are hard to instantiate for most real-world applications. Moreover, trusted parties are hard to find in real-world deployments. Without the aid of a trusted third party, many SGX-based applications mitigate rollback attacks by using TPM's monotonic counters. However, even if rollback attacks are not feasible, an adversary can still clone the victim application in order to mount a forking attack.

3.2 Cloning Attacks in the Wild.

We analyzed the security of 72 SGX-based applications against rollback and cloning attacks. Selected applications were taken from curated lists of SGX papers [13, 24]. We analyzed the application source-code when available; otherwise we analyzed the description provided in the paper where the proposal was introduced.

Our results are summarized in Table 1. Based on our findings, we draw the following observations:

- Out of the 72 proposals, 14 applications (i.e., roughly 20%) are vulnerable to forking attacks based on cloning.
- 11 of the vulnerable 14 applications do not account for any protection mechanism against forking attacks.

```
seal_data

Require: Encrypted data $c_O$
  1: $d$ = Decrypt $(c_O)$
  2: $s$ = Seal $(d)$
  3: OCALL_Store $(i, s)$
```

```
run_interpreter

Require: Encrypted request $c_R$
Ensure: Encrypted result $c_{RES}$
  1: $req$ = Decrypt $(c_R)$
  2: $i, f$ = BISGX_main $(req)$
  3:     $s$ = OCALL_DB_get $(i)$
  4: $d$ = Unseal $(s)$
  5: $c_{RES}$ = Encrypt $(f(d))$
```

Figure 2: Pseudocode of the target functions exposed by the enclave as ecalls: seal_data and run_interpreter

- 3 of the 14 vulnerable applications prevent rollback attacks with a monotonic counter; yet, they are vulnerable to forking attacks based on cloning.
- 7 of the 14 vulnerable applications do not seal state, and therefore are not vulnerable to rollback attacks per design; however, those applications are vulnerable to cloning.
- Out of the 72 proposals, 11 use a TTP to prevent forking attacks. Among these 11 proposals, 9 rely on a decentralized ledger to prevent forking (8 of those are blockchain applications). Finally, 2 applications dismiss rollback attacks by claiming that these attacks can be mitigated by ROTE [131].

We categorize the 14 vulnerable applications in three different categories, namely, A, B, and C. Category A mostly consists of in-memory key-value stores (KVS); by cloning the application, the adversary can split the inputs from different clients across multiple KVS instances so that clients have different "views" of the store (e.g, updates made by one client to a specific key are not seen by another client). Applications in category B seal state to have it available across restarts; by cloning these applications, the adversary can obtain multiple valid states that can be fed to the enclave when it restarts. Category C mostly consists of applications that leverage an SGX enclave as a proxy to guarantee unlinkability or privacy of client requests; by cloning the application, the adversary can partition the set of clients, thereby reducing the anonymity set for each of the clients. We detail how to mount cloning attacks for each category in Appendix E.

3.3 Case Study: Cloning attack against BI-SGX

```
seal_data

Require: Encrypted data $c_O$
  1: $d$ = Decrypt $(c_O)$
  2: Increment(MC)
  3: Read(MC)
  4: $s$ = Seal $(d, \mathrm{MC})$
  5: OCALL_Store $(i, s)$
```

```
run_interpreter

Require: Encrypted request $c_R$
Ensure: Encrypted result $c_{RES}$
  1: $req$ = Decrypt $(c_R)$
  2: $i, f$ = BISGX_main $(req)$
  3:     $s$ = OCALL_DB_get $(i)$
  4: $(d, \mathrm{MC})$ = Unseal $(s)$
  5: if $i == \mathrm{MC}$ then
  6:         $c_{RES}$ = Encrypt $(f(d))$
```

Figure 3: Pseudocode of the patched functions from Figure 2 using monotonic counters. Changes are highlighted in gray.

As a case-study, we show how to successfully mount a forking attack based on cloning against **BI-SGX** [153][2]. We chose BI-SGX because (i) its code is open-source, (ii) it was shown to be vulnerable to forking attacks based on rollbacks and a fix based on monotonic counters was proposed [107]. Our attack against BI-SGX shows

[2]https://github.com/hello31337/BI-SGX

that even if applications use monotonic counters to mitigate forking attacks based on rollbacks, they are still vulnerable to forking attacks based on cloning.

Overview of BI-SGX. BI-SGX provides secure computation over private data in the cloud by leveraging SGX. As shown in Figure 1, a *data-owner* sends to the BI-SGX enclave data `d` encrypted; the encryption key is agreed between the enclave and the data owner via remote attestation. The BI-SGX enclave decrypts the plaintext, seals it, and sends the sealed data (denoted as `s`) to an external database. The database stores `s` along with an index `i` as a tuple `[i, s]`. Later on, a *researcher* can send requests to the enclave; requests include the index that is used to retrieve data from the database and a description of a function `f` to be computed over the data. More precisely, a request includes a tuple `[i,f]`; communication is secured with keys agreed between the enclave and the researcher via remote attestation. Once the enclave receives the request, if `[i,s]` exists in the database, the enclave unseals `s` to recover data `d` and returns `f(d)`. Note that the database lies outside of the enclave boundaries. Therefore, it can be under the control of a malicious OS or cloud provider.

Rollback Attacks on the early version of BI-SGX. A system like BI-SGX should offer some state continuity guarantees. More precisely, as stated by Jangid et al., [107], researcher queries containing different indexes should retrieve and process different data items or, the other way around, queries containing the same index should process the same data item. Jangid et al., [107] used the Tamarin prover to show that BI-SGX could not guarantee such property. Namely, an attacker could feed the enclave with different data even if researchers submit requests with the same index.

To understand how the attack works, we show in Figure 2 the pseudocode for the two main functions manipulating the data from the data-owners and researchers perspective, i.e., `seal_data` and `run_interpreter`, respectively. Note that function `seal_data` does not include the index used for data retrieval; the latter is added by the database when it receives the encrypted data for storage. It is straightforward to see how, upon request issued by the BI-SGX enclave to retrieve data item with index `i`, a malicious OS could return any sealed data item; the enclave has no means to tell if the sealed data returned by the OS is the right one.

Protecting BI-SGX with Monotonic Counters. The aforementioned vulnerability was reported to the developers of BI-SGX by Jangid et al., [107]. The latter also proposed to use monotonic counters (MC) to mitigate this attack. The idea is to seal the index of the data along with the data itself. Hence, when the BI-SGX enclaves requests sealed data with index `i` and obtains a ciphertext `Enc(d,j)`, it only accepts `d` as valid if `i=j`. Further, the use of monotonic counters as indexes ensure that not two data items can be stored with the same index. We implemented the fix suggested by [107] as shown in Figure 3. Here, we use the de-facto "inc-then store" mode of monotonic counters to provide security against rollback attacks.

Forking the "fixed" version of BI-SGX. We argue that this fix is not enough to prevent forks for the BI-SGX enclave. Namely, if there are clones of the enclave running on the system, it is possible to assign the same index to multiple data items. Therefore, when the BI-SGX requests sealed data from the OS, the latter can return one of many valid data items. To carry out this attack, the attacker has to focus on the *data owner* function, i.e. `seal_data`. The process is sketched in Figure 4. The attacker controlling the execution of two BI-SGX enclaves, E and E', has to make sure that both execute **Increment(MC)** before allowing them to proceed with **Read(MC)** (Figure 3). In a nutshell:

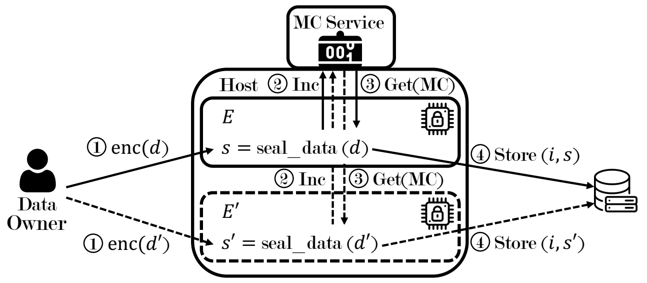


Figure 4: Overview of a cloning attack against the fixed version of BI-SGX that uses monotonic counters.

(1) The adversary starts two BI-SGX enclave instances.
(2) The adversary feeds one data item `d` to enclave E and another data item `d'` to enclave E' (as per figure 4). The current value of the counter is MC (cf. Figure 4 stage 1).
(3) The adversary stops the instance that first executes **Increment(MC)** until the other one has also executed it. The counter at this state is equal to MC+2. For this proof of concept, we have manually synchronized the execution of both instances, in practice an attacker could use a framework such as SGX-Step [175] (cf. Figure 4 stage 2).
(4) The adversary allows both instances to proceed. They execute **Read(MC)** and get exactly the same value of the counter (MC+2) (cf. Figure 4 stage 3).
(5) Instance E seals `(d,MC+2)` while instance E' seals `(d',MC+2)`. Both ciphertexts are sent to the database. Both ciphertexts are valid for a query from a *researcher* to process data stored at index MC+2, as the BI-SGX enclave only checks if MC in the sealed blob is equal to the index value in the *researcher* request (cf. Figure 4 stage 4).

We note that the adversary is not limited by the number of instances that can be launched at the same time.

We responsibly disclosed this vulnerability to the developers of BI-SGX. They agreed to take into account attacks based on cloning for further releases of BI-SGX. In Section 7, we show how our proposed solution, CloneBuster, can efficiently detect any enclave cloning attempts in between the execution of **Increment(MC)** and the data sealing phase.

4 CLONEBUSTER

4.1 System & Threat Model

Given the observations made in Section 2 and in Section 3, we focus on the practical problem of detecting clones on a single platform, in realistic application scenarios where the OS is malicious and the enclave has no access to a trusted third party. As shown in Table 1, such a setting faithfully mimics most existing deployments.

We consider two enclaves to be clones if (i) they have been loaded with the same binary (hence, they have the same MRSIGNER

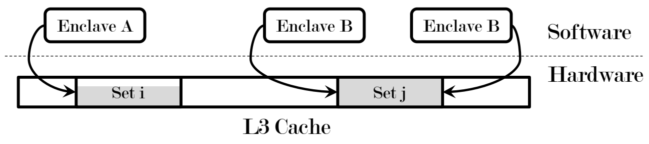


Figure 5: Signaling mechanism of CloneBuster. Each enclave uses a group of L3 cache sets to signal its presence and detect the presence of clones. Enclaves with the same (resp. different) binary use the same (resp. different) cache sets.

and MRENCLAVE)[3], and (ii) they run at the same time. Condition (i) also implies that clones of an enclave share long-term public keys; condition (ii) is necessary for a successful forking attack as explained in the previous section.

We assume the common threat model for Intel SGX where the hardware is part of the TCB, but the adversary controls privileged software (e.g., the OS) on the host. The goal of our adversary is to run multiple clones on a platform while bypassing the detection mechanism. Similar to [52, 66, 68, 69, 91, 119, 164, 168, 171], we consider Denial of Service (DoS) attacks to be out of scope. We note that a malicious OS can anyway DoS a process running on its platform—irrespective of the defense mechanism employed.

4.2 Overview of CloneBuster

We seek to design and build a detection mechanism that (i) can be deployed without relying on external trusted parties, (ii) cannot be escaped by a malicious OS despite its privileges, and (iii) remains robust despite other benign applications running on the same platforms.

The main intuition behind CloneBuster is to rely on a covert channel as a signaling mechanism so that each enclave can indicate its presence to (and detect the presence of) clones. Namely, if the enclave instance is truly unique, it will see no response on the channel being monitored. On the other hand, if multiple instances are running, each instance will observe a measurable response in the form of a contention pattern. The challenges in using a covert channel as a signaling mechanism for clones lie in how to make communication robust despite (benign) noise due to other applications on the platform and, most importantly, despite a malicious OS that may tamper with the channel so that two clones do not detect each other.

CloneBuster leverages a cache-based covert channel [128] to detect clones since such a channel does not require clones to be synchronized, i.e. data persist even after the execution of the process has finished, and works even if they run on different cores (by using L3 cache). We note that covert channels could also leverage the TLB [176] or the DRAM [142]. However, the a covert channel based on TLB only works between two hyperthreads, whereas a DRAM-based covert channel only works if the communicating parties access the same memory bank simultaneously.

We define the channel as a specific group of cache sets. CloneBuster requires the enclave to fill these cache sets with its own data, and to continuously measure the time to access such data, in order to detect if it is still cached or if it has been evicted. Clones will use the same channel (i.e., the same group of cache sets), removing each other's data. Thus, the sequence of cache hits and misses allows to distinguish whether clones are running on the same host. This process is depicted in Figure 5.

Data that resides in the cache is accessed faster (cache hits) than data that is not placed in cache (cache misses). In order to distinguish cache hits from misses, a process needs to measure data access times. Intel processors expose timers through the *rdtsc* and *rdtscp* instructions that provide enough resolution to carry out these measurements. Unfortunately, these instructions are not available inside an enclave. As shown in [126, 160], it is possible, however, to leverage a counting thread as an alternative that provides even higher resolution compared to the *rdtsc* instruction. In CloneBuster, we opt for this approach.

Given a sequence of hits and misses, a detection algorithm determines whether a clone enclave is running on the host. This information may be used by the enclave application to, e.g., halt execution and/or signal the event to an external party (i.e., the enclave developer).

We show in Section 5 that a malicious OS cannot prevent two clones from detecting each other because (i) the OS has limited choice on which cache-based channel to assign to a given enclave, and (ii) CloneBuster ensures that an enclave monitors all the relevant cache-based channels to ensure reliable detection.

In the following, we provide details on how CloneBuster selects the cache sets to be monitored, and how it builds and uses the eviction sets to monitor the cache sets.

4.3 Channel Selection

CloneBuster uses the cache as a channel for an enclave to signal its presence to (and detect the presence of) other enclaves with the same binary. Detection succeeds as long as enclaves with the same binary monitor the same channel, and enclaves with different binaries monitor different channels.

Assuming a typical cache with s slices and 1024 sets per slice, there are 10 bits of a physical address that determine the cache set index (bits 6-15). An enclave only manages 6 of those bits (6-11), but it is unaware of the remaining 4 bits (12-15) that are controlled by the OS. By fixing bits 6-11 of an address, the enclave reduces the possible cache sets where a block of data is being cached within a slice to 16. If all enclaves loaded with the same binary monitor the same 16 cache sets determined by a specific value of bits 6-11, each of them can detect the presence of its clones—despite an adversary that controls the OS and allocates the physical pages of the enclave. We provide more details on cache memories and how cache-based covert channels work in Appendix A.1.

Therefore, CloneBuster defines a channel as a group of 16 cache sets, in principle allowing for up to 64 concurrent channels. In Appendix B, we show that this choice is optimal, since monitoring less than 16 sets may allow the OS to execute multiple clones of an enclave and evade detection. Note, however, that the channel selected by a given enclave (e.g., by fixing bits 6-11 of the addresses to be monitored) must not be secret and, in particular, security is not affected if the OS knows which channel is being used by an enclave. In a real-world deployment, the OS may even actively help enclave owners in selecting an unused channel prior to attestation; in turn, the enclave owner uses attestation and secret provisioning

[3]This also means that each enclave can access data sealed by its clone.

Algorithm 1 Building the eviction sets in CloneBuster

```
Require: Memory byte array memArr[24MB];
Ensure: evictionSets[16][SLICES]
 1: $spoilerArr[256][LIM] \leftarrow \{\}$                         ▷ LIM depends on memArr size
 2: $cacheGroups[16][16 * LIM] \leftarrow \{\}$
 3: $evictionSets[16][SLICES * WAYS] \leftarrow \{\}$
 4: for $i = 0$ to 256 do
 5:     $cont \leftarrow 0$
 6:     test_address = memArr[i*PAGE_SIZE + offset];
 7:     spoilerArr[i][cont++] = test_address;
 8:     for $j = (i + 1)$ to 24MB; j+=PAGE_SIZE; do
 9:         if aliasing(test_address,memArr[j*PAGE_SIZE]) then
10:             spoilerArr[i][cont++] = memArr[j*PAGE_SIZE];
11: // Group the addresses with same set number
12: for $i = 0$ to 16 do                                      ▷ Reduce before expand
13:     $cont \leftarrow 0$                                    ▷ it is 16 at the end of each iteration
14:     test_array = spoilerArr[i][:];
15:     cacheGroups[i][cont++] = test_array;
16:     // Remove used data from the copy array
17:     spoilerArrCopy ← (spoilerArr - cacheGroups)
18:     for $j = i + 1$ to 256 do
19:         remove spoilerArr[j][:] from spoilerArrCopy;
20:         if test_array is not evicted by spoilerArrCopy then
21:             cacheGroups[i][cont++] = spoilerArr[j][:];
22:             write spoilerArr[j][:] back at spoilerArrCopy;
23:     for $j = 16$ to 256 do                                ▷ Find remaining groups
24:         test_array = spoilerArr[j][:];
25:         if test_array is evicted by cacheGroups[i][:] then
26:             cacheGroups[i][cont++] = test_array;
27: for $i = 0$ to 16 do
28:     $evictionSets[i][:] = \mathbf{reduce}(cacheGroups[i][:])$
```

to instruct the enclave about which channel to use. Note that the OS has no advantage in assigning two different enclaves—loaded with different binaries—to the same channel as this leads to a DoS. In this case, the two enclaves will (mistakenly) detect a clone and take appropriate countermeasures (e.g., stop their execution or report the problem to an external party like the enclave owner). In practice, a malicious OS can easily DoS a process running on its platform—regardless of whether CloneBuster is used or not.

4.4 Building Eviction Sets

Popular techniques to build eviction sets from within an enclave [160] require that the OS assigns contiguous memory to enclaves. In our settings, a malicious OS may, however, assign non-contiguous memory to the enclave. Therefore, we leverage alternative techniques that rely on false dependencies on load operations which are not under direct control of the OS [105]. We provide additional details on our choice in Appendix B; more specifically, we use the SATisPy [89] SAT solver to show that a malicious OS may evade detection if evictions sets are built relying on the assumption that enclave memory is contiguous.

We leverage the technique of [105] to group data whose physical addresses share the last 20 bits and then regroup that data into groups that share the last 16 bits (i.e. groups that share the cache set number). Since 12 out of these 20 bits are controlled by the enclave, we can create $2^8 = 256$ different groups that we call "spoiler groups". This step, in turn, ensures that we have enough distinct addresses to build the necessary eviction sets. The spoiler groups are then regrouped into cache groups, and finally, cache groups are reduced and arranged so that all the slices are covered.

The process is summarized in Algorithm 1. We use an array of 24MB—twice the size of our cache memory—so to ensure that all possible eviction sets can be built. We also point out that when building the "spoiler groups" it should be verified that the *test_address* (line 6) is not already present in the *spoilerArr*. Similarly, the *test_array* should not be part of the *cacheGroups* (line 14). These checks have been omitted in the pseudo-code for simplicity and brevity.

The *cacheGroups* array is filled in two stages. In the first stage (loop at line 18), a group of arrays or a group of addresses with the same set number that can occupy all the respective slices is obtained. At this point, the data in the *cacheGroups* could be re-arranged per slices and then reduced to its minimum core (i.e. it should include as many addresses as ways of the cache sets), which is the goal of this algorithm. That is, one could directly execute the steps at line 27. On the other hand, the second stage (lines 23-26) ensures that the OS has assigned to the enclave the $2^8 = 256$ addresses corresponding to the aforementioned 8 bits of a "spoiler address". Besides, the distances between addresses included in the *spoilerArr* and between the indexes of each *cacheGroup* show if the memory assigned by the OS is linear and if there are any gaps, i.e., unassigned pages.

We note that by having 256 different groups of addresses, we ensure that all the possible set numbers are covered. Moreover, by re-grouping those 256 groups into the 16 groups that share the same cache number while ensuring all the cache slices are covered, we guarantee that CloneBuster could map any cache location. In case any of the tests fail, this offers compelling evidence that the OS is manipulating memory to alter the expected view of the memory by CloneBuster—in this case, the enclave should refuse to execute. It is worth noting that the value of the offset used in line 6 is chosen so that the virtual address of $memArr[offset]$ has its bits 6-11 equal to the selected channel, if the number of cores is not a power of two due to its slice selection function [104, 189]. If, on the contrary, the number of cores is a power of two, the aforementioned slice selection function [135] makes it possible to use any value for the $offset$, but it should be changed afterwards (e.g during the reduction phase). Finally, the algorithm used to obtain the minimum-size eviction sets from a bigger set of addresses mapping to the same set (*cacheGroups*), could be any of the ones proposed in the literature, e.g., [128, 140], that mainly remove elements from the array until it has the same size as ways of the cache, while ensuring it is still able to completely fill the set. In practice, we have taken an approach similar to [128].

We believe that our technique to build eviction sets may be of general interest as it provides a robust means to build eviction sets inside SGX enclaves.

4.5 Monitoring Phase

During the monitoring phase, CloneBuster reads the data of the sets to be monitored in a loop. Namely, CloneBuster measures the access times to each of the data blocks in the sets, in order to determine whether they are still cached (hit) or not (miss). The sequences of cache hits or misses—that we refer to as "observation windows"—are fed to a classification algorithm that decides whether a clone is running on the same host. Like in [160], we leverage a counting thread to measure access time: we fetch the value of the

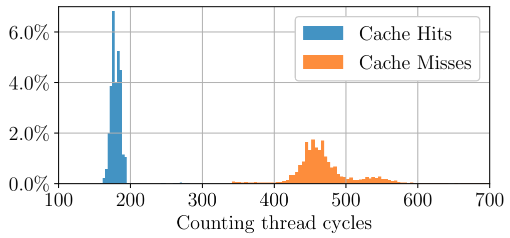


Figure 6: Times distribution of cache hits and misses in our test machine measured with the counting thread.

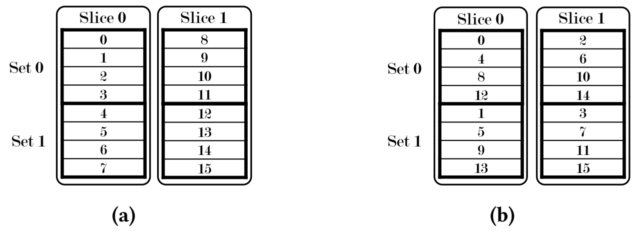


Figure 7: Different access patterns to the data included in the monitoring set for a 4-Way set associative cache with 2 slices and 2 sets per slice.

counter before and after reading an address. If the difference of the two counter values is greater than a pre-defined threshold, we conclude that the data was not cached and treats it as a cache miss; otherwise, we assume a cache hit.

The threshold to distinguish cache hits from misses is machine dependent; it can be pre-computed if the hardware where the enclave is deployed is known a priori. Otherwise, the monitoring thread can compute the threshold by flushing and reloading a block of data (cache miss time), reading again that block of data which will be in the cache (cache hit time), and repeating this process while computing the mean times. As shown in Figure 6, cache misses take significant longer than cache hits and can be easily distinguished by observing data access times.

Note that the monitoring and counting threads should run continuously, whenever the enclave is executing a critical piece of code where no clones must be allowed (e.g., between a read and an increment of a monotonic counter). If the monitoring/counting thread is interrupted, the obtained measurements will not match the expected ones, i.e., they will differ from the distributions depicted in Figure 6. We treat such an event as evidence that the OS is manipulating the enclave with malicious intent and take countermeasures (e.g., halt the execution of the enclave).

Note that before the monitoring phase can actually start, the enclave has to pre-fetch the data to be monitored into the cache to ensure that all the observed cache misses are due to evictions caused by other processes.

We point out that there is no need for an enclave to fill all the ways of the monitored cache sets. In particular, given a W-way set-associative cache, clones will evict from cache each other's data—hence, will detect each other—as long as the number of ways filled per cache set, namely m, is chosen such that $(W/2) < m \leq W$. Further, if $m = W$, the enclave may detect evictions due to benign applications that happen to use the same cache sets and output a false positive.

Another important design choice that affects the performance of CloneBuster is the order in which the enclave loops over the addresses in the monitoring group. In particular, the order impacts the latency in detecting clones. This is exemplified in Figure 7. Here, we depict a cache with two slices and two sets of 4 ways each. The numbers on each line represents the order in which the enclaves touches the lines in a loop iteration. Assuming the pseudo-LRU eviction policy implemented in Intel processors [62, 178], data stored at lines touched earlier becomes the candidate to be evicted in case of conflict. If the enclave accesses all lines of a set before moving to the next one (Figure 7a), it might not experience cache misses—thus may not detect a clone—before it loops over all of the address of the monitoring group. On the contrary, if the enclave accesses one line per set and then moves to the next set (Figure 7b), in the worst case scenario, it will start observing cache misses after looping over one line in each of the sets. In other words, if m stands for the number of ways filled, c for the number of sets in the channel and s for the number of slices in the cache, accessing all the data in the set before moving to the next one might require up to $m * c * s$ accesses prior to detection, whereas accessing one line per set can take up to $c * s$ accesses. In the sequel, we therefore adopt the latter access pattern.

5 SECURITY ANALYSIS

Recall that the goal of the adversary is to execute two (or multiple) clones—enclaves loaded with the same binary—while evading the clone-detection mechanism.

One possible attack strategy is to assign two different channels (i.e., two different groups of cache sets) to two enclave clones. We eliminate this option by ensuring that any two enclaves, loaded with the same binary, monitor the same group of cache sets. In particular, if all enclaves with the same binary fix bits 6-11 of the addresses to be monitored, each of those addresses can only be mapped to one out of 16 cache sets. By monitoring all of the 16 cache sets, we guarantee that two clones cannot be assigned to different channels. Note that monitoring less than 16 cache sets—out of those determined by fixing bits 6-11 of an address—may allow the adversary to evade the detection mechanism. In particular, we used a SAT-solver (SATisPy [89], which in turn is a wrapper of MiniSAT [82]) to simulate memory mapping and to show that, if less than 16 cache sets are monitored, the OS can find multiple mappings that effectively assign clones to different channels. We provide more details on this in Appendix B.

Alternatively, the adversary might leverage the ability to control the execution of the enclave at instruction level, e.g., by using frameworks such as SGX-Step [175]. By choosing which of the clones is making progress, one or few instructions at a time, the adversary may prevent one enclave instance from detecting the presence of the other. We argue that such strategy is not viable because the cache as a covert channel allows two enclaves to detect each other, even if they are not running at the same time. Take, for example, the BI-SGX enclave described in Section 3. The enclave uses monotonic counters and a forking attack requires two clones, say E and E', such that the following instructions are executed in a sequence: (a) E calls `Increment(MC)`, (b) E' calls `Increment(MC)`, (c) E calls `Read(MC)`, and finally (d) E' calls `Read(MC)`. The outcome is two sealed data items, one from E and the other from E', with the same value of the monotonic counter. This attack can be mitigated by using CloneBuster. In particular, if E runs first, it writes its fingerprint to the cache. Next E' runs and overwrites with its own fingerprint what enclave E had written into the cache. Finally, E resumes, detects that its fingerprint into the cache was overwritten and determines that a clone is running. Once a clone has been detected, the enclave could take appropriate countermeasures (e.g., refuse to seal data).

Note that "polluting" the cache-based channel is not a viable option for a malicious OS. If the OS deliberately touches the cache lines used by CloneBuster, the detection mechanism (wrongly) infers that a clone is running thereby generating a false positive. Upon detection, the enclave may, e.g., halt its execution but no fork would take place.

Further, the OS may as well try to make a cache miss look like a hit so that the enclave running CloneBuster fails to detect its clone. To do so, a malicious OS needs to slow down the counting thread while the monitoring thread measures access times to its eviction set. Note that the OS cannot selectively shut down threads of an enclave. It may prevent a thread (and in particular, the counting thread) from making progress by scheduling it on a core along with other applications. This occurrence will be immediately detected by CloneBuster as the monitoring thread will experience a sequence of counter reads with no increment (while the core of the counting thread is busy with the other application). In this case, CloneBuster would halt the execution and, e.g., inform the enclave developers.

Another approach to make a cache miss look like a hit would be to change the frequency of the different cores available. Concretely, the adversary may run the counting thread on a slower core and the monitoring thread on a faster one. We note that there is no SGX-enabled processor with per-core frequency scaling; this feature seems to be available only on some HPC processors that do not feature SGX [93, 157]. Hence, if the OS changes the frequency of a core in an SGX-capable processor, it would cause a frequency change on all other cores [148, 166]. We have empirically verified this in our platform. Even assuming future processors with SGX and per-core frequency scaling [101], some time elapses between the instant when the OS makes a frequency change request until this change is effective. As reported in [93, 157], this time interval amounts to roughly 500 μs; in contrast a cache miss only takes around 0.15 μs. Thus, adding a periodic re-calibration phase where the monitoring thread measures the time of a cache miss, may prevent the OS from scaling the frequency of single cores. In particular, if the re-calibration phase occurs every 500 μs, frequency scaling by the OS could be spotted. As an alternative strategy, the OS may configure core frequencies in advance, and then move the counting thread across cores. Again this can be spotted with periodic re-calibration.

6 OTHER APPLICATIONS OF CLONEBUSTER

6.1 Extending CloneBuster to detect $N > 1$ Clones

So far, we have described how CloneBuster is used to allow the execution of a single legitimate instance of an application enclave, and to detect whenever the number of cloned instances is two or greater. In what follows, we show how CloneBuster can allow the execution of $N \geq 2$ legitimate instances and detect whenever the number of instances reaches $N + 1$ or above. This could be useful in application scenarios where more than one enclave instance may be honestly needed for redundancy or increased throughput [165].

As mentioned earlier, allowing $N = 1$ instances requires that an enclave monitors at least $W/2 + 1$ of the available ways in a set. To generalize to $N \geq 2$, we tune the number m of monitored ways in a set such that (i) as long as N clones are running, none of them evicts data pushed to cache by the other instances, and (ii) if $N + 1$ clones are running, the enclaves will witness evictions of their data from the cache. In other words, we set m such that $m \leq W/N$ and $m > W/(N + 1)$. For example, if the number of ways in a cache set is $W = 16$ and we wish to allow up to $N = 2$ concurrent instances, we set $m = 6$. As a result, two instances will fill at most 12 ways of a cache set, whereas a third instance will fill up all the lines and evict data of the first two. Assuming a 16-way last-level cache, Table 2 shows possible values of m for the different values of N. Note that for some values of N, no m satisfies the constraints defined above. For example, no integer m satisfies $W/(N + 1) < m \leq W/N$ for $W = 16$ and $N = 7$.

As a by-product, we argue that CloneBuster can also estimate the number of clones running on a machine by dynamically tuning the number of monitored ways per cache set as shown in Table 2. For example, the enclave starts monitoring $m = 12$ ways per set to check for the presence of a clone; in case the classifier infers the presence of another instance, the enclave decreases m to 8 to spot whether two or more other instances are running. If, given the new value of m, the classifier outputs that there are no clones, the enclave concludes there was only one other instance running in the system. Otherwise, if the classifier infers again the presence of clones, the enclave decreases again m to 5. Again, if the classifier outputs that there are no clones, the enclave concludes there were only two instances running in the system. On the contrary, if the classifier keeps detecting clones, the enclave decreases m once again. This process is repeated until no clones are detected, which would then allow the enclaves to estimate the number of clones running on the platform.

6.2 Using CloneBuster to Enhance NARRATOR and ROTE

As mentioned earlier, solutions against forking attacks like ROTE [131] and NARRATOR [139] use a cohort of enclaves hosted in different platforms that act as a (distributed) TTP and provide forking prevention to other application enclaves. Both solutions require yet another TTP to avoid being forked at system initialization, since a fork on the cohort allows the adversary to fork application enclaves.

ROTE [131] uses a trusted administrator that provisions secret keys to the cohort members; it uses the linkable attestation mechanism of Intel SGX to ensure that at most one enclave per platform is provisioned. On the other hand, NARRATOR [139] replaces the centralized administrator with a distributed TTP, i.e., a blockchain. In particular, enclaves use the hash of their sealing key (derived from the platform private key and the enclave identity) as a platform ID. An enclave being initialized as a cohort member checks if

# of allowed instances (N)	1	2	3	4	5	8	16
# of ways to be monitored (m)	9-16	6-8	5	4	3	2	1

Table 2: Number of ways that have to be monitored per cache set to allow different numbers of legitimate enclaves, given a 16-way set associative cache.

its platform ID has been written on the ledger; if not, the enclave writes the ID to the ledger and joins the cohort. Otherwise, the enclave assumes that a clone on the same platform has already joined the cohort and refuses to proceed.

CloneBuster can be leveraged to remove the need of a TTP (be it a trusted administrator or a blockchain) during the initialization phases of ROTE or NARRATOR. In a nutshell, an enclave being initialized to join the cohort uses CloneBuster to ensure that no clone on the same platform is also being initialized.

7 IMPLEMENTATION AND EVALUATION

7.1 Implementation Setup

We implemented a prototype of CloneBuster, including the code for the creation of the eviction sets (and some tests to ensure they have been properly built) and the counting thread that serves as a clock. Our implementation accounts for approximately 800 LoC.

We deployed the prototype on a Xeon E-2176G (12 vCores at 3.70GHz, 64 GB RAM, and a 12 MB 16-way cache). To assess the performance of CloneBuster, we evaluated the impact on performance of (i) the choice of classification algorithm used to infer the presence of a clone given a sequence of cache hits and misses, (ii) the number of ways per set to be monitored m, and (iii) the size of the observation window w.

We evaluate performance in an ideal scenario where no other application apart from the enclave (and possibly its clone) is running, as well as in a more realistic scenario where background processes—taken from the Phoronix benchmark suite [120]—are running on the host at the same time. In scenarios featuring background processes, we run as many instances of the benchmark as needed to reach a total CPU usage close to 100%. For each configuration of parameters and background process, we collected 100.000 samples while the enclave and a clone are running, and the same number of samples while the enclave is running without clones. We labeled these samples accordingly and obtained multiple datasets of 200.000 samples per scenario.

7.2 Evaluation Results

We assess the performance of CloneBuster by means of F1 score. Each data point represents the mean of 10-fold cross-validation.

Choice of the Detection Algorithm. We evaluate the performance of different detection algorithms in inferring the presence of a clone, given a sequence of cache hits and misses. In particular, we considered a number of classifiers included in Scikit-learn [141] as well as a simple threshold-based algorithm. For the latter, the threshold t of cache misses for the detector to report a clone is selected empirically, as the one that allows to obtain the best performance.

Figure 8 compares the performance of various detection algorithms for $w \in [1, 2024]$ and for $m \in \{9, 12, 16\}$ both in the ideal scenario with no background processes and in a realistic scenario where processes are running in background. For the latter scenario, we use `x265 video encoder`—the application with the most intensive memory use among the ones we have tested from the Phoronix benchmark suite—as the background process.

The comparison between the plots on the left side of Figure 8 (with no background process) and the ones on the right side of the same figure (with `x265 video encoder` running in parallel) allows us to assess the impact of background processes on the performance.

We note that the threshold-based algorithm is among the ones with higher F1 scores, for most configuration of w and m. Indeed, the threshold-based detector emerges as the most suitable choice—owing to its simplicity, small code-size, and F1 score.

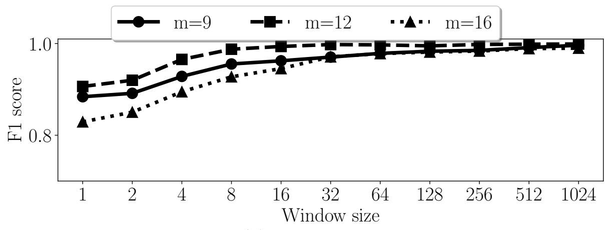
Additional results with different background applications are summarized in Table 3. Furthermore, in Appendix C, we provide additional results with alternative detection algorithms and background applications of the Phoronix benchmark suite.

Impact of observation window size w. Figure 8 also shows the impact of the size of the observation window w on the F1 score. Clearly, increasing w leads to better performance. In particular, a small observation window may only account for cache misses due to benign applications running on the same host, and may cause false positives. For example, by using the threshold-based classifier with $w = 1$, the F1 score for $m = 9$, $m = 12$, and $m = 16$ is 0.884, 0.906, and 0.829, respectively in an ideal scenario; when `x265 video encoder` is running in the background, F1 scores are 0.801, 0.907, and 0.757 for $m = 9$, $m = 12$, and $m = 16$, respectively. By increasing w, classification becomes more robust: with $w = 1024$, F1 score is 0.996 ($m = 9$), 0.999 ($m = 12$), and 0.990 ($m = 16$) in the scenario where no application is running in the background and reaches 0.999 ($m = 9$), 0.994 ($m = 12$), and 0.982 ($m = 16$) when `x265 video encoder` runs in the background.

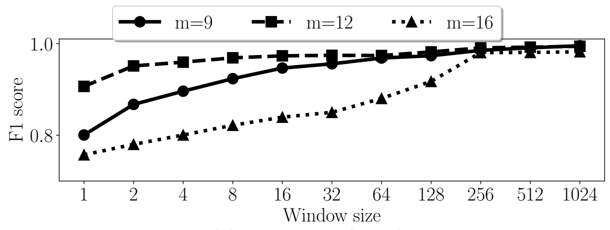
We also note that w has a direct impact on detection latency, since it determines the time to fill the observation window with cache hits and misses—before the window is fed to the classifier. For instance, given that measuring a cache miss on the test machine takes approximately 450 cycles, setting $w = 256$ results in detection latency of roughly 115k cycles. To put this number in context, computing an RSA-2048 signature with `openssl` on the test machine requires 17390k cycles.

Impact of number of ways m. As expected, the detector performance in a scenario with no background processes is only marginally affected by m—because no other process is polluting the cache. In scenarios with background processes, the impact of m on the F1 score is more prominent since those processes may be polluting the monitored lines and may be causing false positives.

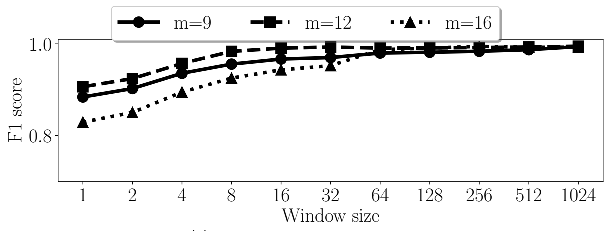
Indeed, $m = 16$ resulted in the highest number of false positives for most of the configurations tested. On the other hand, performance difference between $m = 9$ and $m = 12$, depends on the process running in the background. We observe a slight imbalance between precision and recall among the two cases. While $m = 12$ features higher recall, $m = 9$ obtains better precision. In other words, the number of false positives is higher for $m = 12$ and the number of false negatives is higher for $m = 9$. Since we consider false negatives more damaging than false positives, we opt for $m = 12$.



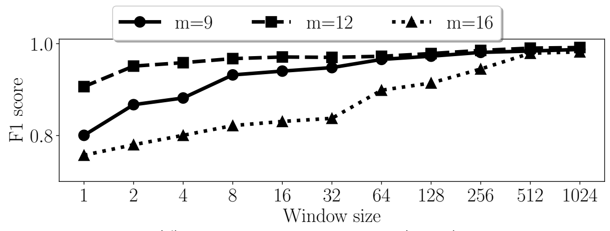
(a) Threshold



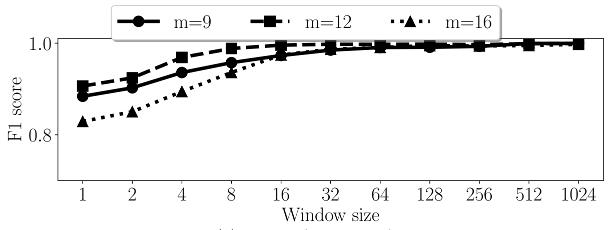
(b) Threshold (x265)



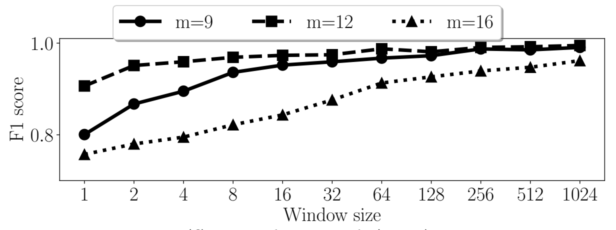
(c) Gaussian Naive Bayes



(d) Gaussian Naive Bayes (x265)



(e) Neural Network



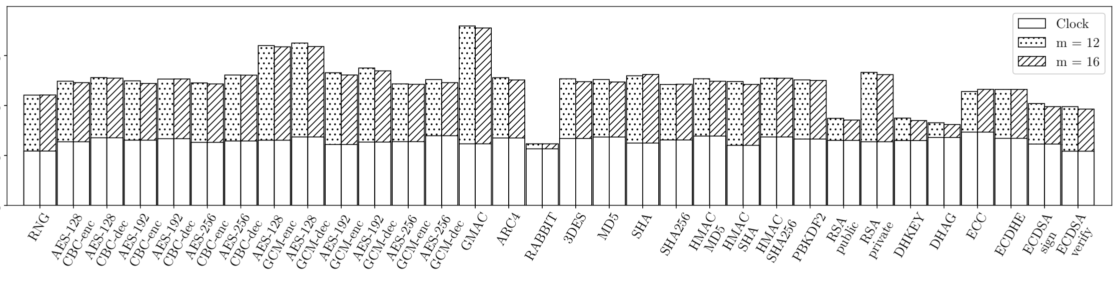
(f) Neural Network (x265)

Figure 8: F1 score of various detection algorithms for different values of w and m. Figures on the left show the performance of CloneBuster with no other application; figures on the right show the performance when `x265 video encoder` runs in the background.

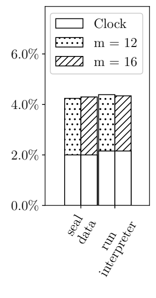
		Threshold						**Naive Bayes**						**Neural Network**					
		w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024
Baseline	m=9	0.884	0.928	0.962	0.979	0.985	0.996	0.884	0.936	0.967	0.980	0.984	0.994	0.884	0.936	0.973	0.991	0.993	0.999
	m=12	0.906	0.965	0.994	0.997	0.998	0.999	0.906	0.957	0.991	0.991	0.992	0.995	0.906	0.969	0.996	0.997	0.997	0.999
	m=16	0.829	0.895	0.945	0.978	0.984	0.990	0.829	0.895	0.980	0.987	0.995	0.993	0.829	0.894	0.974	0.991	0.994	0.997
x265	m=9	0.801	0.896	0.947	0.998	0.997	0.999	0.801	0.882	0.940	0.967	0.981	0.988	0.801	0.895	0.952	0.967	0.987	0.991
	m=12	0.907	0.959	0.973	0.974	0.991	0.994	0.907	0.959	0.971	0.973	0.985	0.992	0.907	0.959	0.973	0.987	0.991	0.995
	m=16	0.757	0.801	0.839	0.880	0.980	0.982	0.757	0.801	0.831	0.899	0.945	0.982	0.757	0.795	0.844	0.913	0.940	0.962
sql	m=9	0.894	0.914	0.954	0.968	0.990	0.995	0.894	0.919	0.958	0.988	0.992	0.996	0.894	0.922	0.960	0.992	0.995	0.998
	m=12	0.945	0.973	0.978	0.973	0.987	0.992	0.945	0.973	0.977	0.988	0.990	0.995	0.945	0.973	0.978	0.985	0.990	0.994
	m=16	0.812	0.846	0.854	0.902	0.955	0.980	0.812	0.841	0.845	0.875	0.933	0.981	0.812	0.845	0.859	0.913	0.960	0.975
opencv	m=9	0.894	0.922	0.941	0.947	0.965	0.987	0.855	0.889	0.924	0.932	0.955	0.971	0.855	0.888	0.934	0.947	0.971	0.979
	m=12	0.903	0.930	0.937	0.958	0.967	0.984	0.903	0.930	0.936	0.956	0.964	0.984	0.903	0.930	0.938	0.967	0.973	0.987
	m=16	0.670	0.775	0.876	0.883	0.923	0.963	0.670	0.775	0.868	0.888	0.913	0.953	0.670	0.775	0.921	0.932	0.956	0.971
gcc	m=9	0.850	0.909	0.954	0.962	0.972	0.977	0.850	0.914	0.950	0.986	0.983	0.983	0.850	0.914	0.958	0.986	0.983	0.987
	m=12	0.931	0.973	0.980	0.984	0.985	0.991	0.931	0.970	0.976	0.980	0.983	0.992	0.931	0.973	0.980	0.981	0.983	0.989
	m=16	0.809	0.838	0.874	0.908	0.915	0.942	0.809	0.837	0.879	0.902	0.925	0.985	0.809	0.838	0.895	0.917	0.932	0.964
cloud	m=9	0.933	0.958	0.976	0.981	0.988	0.990	0.933	0.967	0.984	0.972	0.986	0.992	0.933	0.967	0.983	0.984	0.993	0.995
	m=12	0.906	0.933	0.953	0.980	0.990	0.993	0.906	0.941	0.957	0.986	0.987	0.991	0.906	0.941	0.956	0.988	0.992	0.992
	m=16	0.856	0.916	0.935	0.981	0.987	0.988	0.856	0.916	0.935	0.946	0.972	0.978	0.856	0.921	0.936	0.948	0.980	0.990

Table 3: F1 score of several detection algorithms for different values of w and m. "Baseline" refers to the scenario where no background application is running, whereas the others refer to different applications running in the background.

Performance overhead for WolfSSL. We use applications of WolfSSL [183]—a suite of cryptographic applications ported to SGX—as exemplary applications to assess the overhead of CloneBuster. For each application in the WolfSSL benchmark, we run the

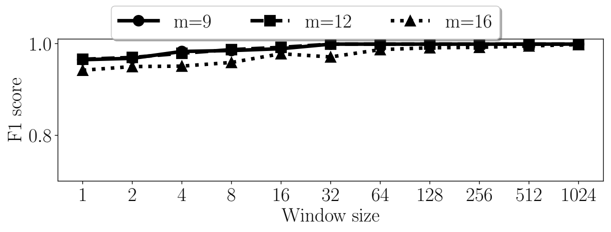


(a) WolfSSL

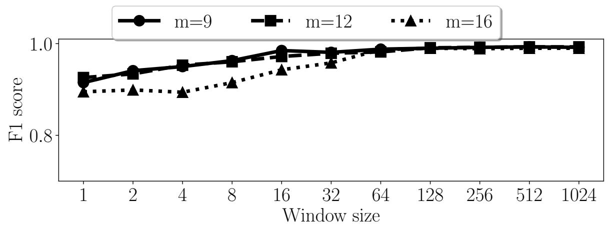


(b) BI-SGX

Figure 9: Mean penalty (in achieved throughput) due to CloneBuster for different WolfSSL applications (a) and BI-SGX (b). Here $w = 32$.



(a) Threshold (no background process



(b) Threshold (x265)

Figure 10: F1 score of the threshold based detection algorithm for different values of w and m for attacks against BI-SGX. Figure on the left show the performance of CloneBuster with no other application; figure on the right show the performance when `x265 video encoder` runs in the background.

vanilla version as baseline and compare its throughput with the one of the same application when enhanced with CloneBuster.

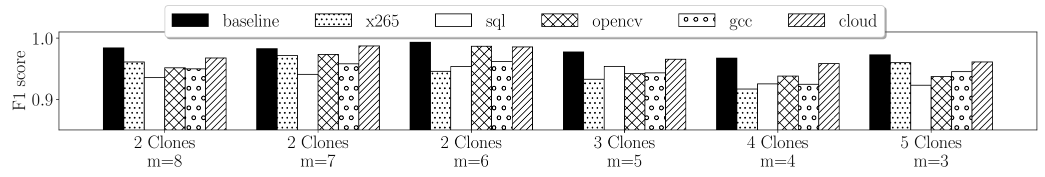
Figure 9 (a) depicts the performance penalty incurred for each application in WolfSSL, normalized with respect to the baseline. Each data point is averaged over 100 independent runs. Here, "clock" refers to the performance of the application instrumented with CloneBuster but with only the counting thread running, whereas "m=12" and "m=16" refer to the performance of the application when both the counting and monitoring threads of CloneBuster are running. The mean performance penalty across all applications of the WolfSSL benchmark is 2.58 ± 0.17 % if just the counting thread is running, and 4.82 ± 0.91% and 4.88 ± 0.90% if the countermeasure is running with 12 and 16 monitored ways, respectively. We conclude that parameter m has little effect on the overhead and that the performance penalty due CloneBuster can be tolerated by most applications.

Evaluating CloneBuster when used with BI-SGX. We now evaluate the performance penalty incurred by BI-SGX [153] when CloneBuster is used to detect attacks. Figure 9 (b) shows the penalty for the two main functions of BI-SGX, namely `seal_data` and `run_interpreter` (Figure 3). We measure the time for each function to execute with input data comprised of 5000 characters. The performance penalty is normalized with respect to the baseline (i.e., BI-SGX without CloneBuster) and we report the average over 100 runs. The mean performance penalty was measured to be 1.99 ± 2.15% if just the counting thread is running, and 4.24 ± 4.39% and 4.30 ± 4.33% if CloneBuster is running and monitoring 12 or 16 ways, respectively. In Figure 10, we asses the performance of CloneBuster in detecting clones of BI-SGX. We use the threshold-based detection algorithms for $w \in [1, 1024]$ and for $m \in \{9, 12, 16\}$, both in the ideal scenario with no background processes and in a realistic scenario where a background process (`x265 video encoder`) runs in the background. We collect samples for 10,000 executions of BI-SGX running in a benign setting and while carrying out the attack described in Section 3.3, respectively. Figure 10 shows that even with background noise, the F1 score reaches 0.999 for $w \geq 64$.

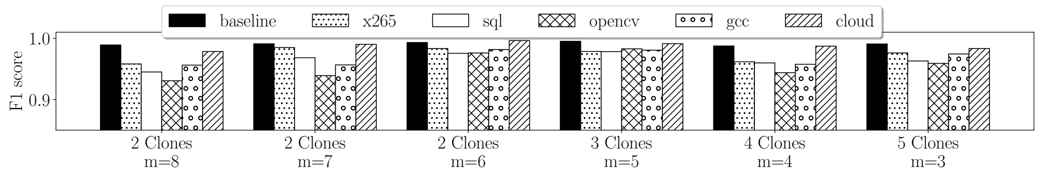
Detection of multiple clones. We now focus on assessing the effectiveness of CloneBuster in detecting malicious clones when $N \geq 2$ legitimate instances are allowed to run and an alarm should be raised as soon as $N + 1$ instances are running. That is, for each value of $N \in \{2, 3, 4, 5\}$ we run either N or $N + 1$ instances and assess whether CloneBuster can distinguish the two scenarios.

Figure 11 shows the result for this set of experiments. The gray-colored bars show the baseline performance, i.e., when no other application is running on the machine, apart from the N enclave instances (and possibly an additional clone); colored bars depict the results when other processes are running in the background. Note that m is chosen according to Table 2 and w is set to either 16, 32, or 256.

In Figure 11a, we set $w = 16$ for $N = 2$ and $w = 32$ for $N \in \{3, 4, 5\}$ because, for $N \in \{3, 4, 5\}$, we witnessed larger increase of the F1 score when doubling the size of the observation window



(a) w=16 for the case of 2 clones and w=32 for the remaining cases



(b) w=256

Figure 11: F1 scores of the threshold-based detection algorithm for the detection of $N + 1 > 2$ clones.

from $w = 16$ to $w = 32$. Concretely, for $N \in \{3, 4, 5\}$, increasing w to 32 improves the F1 scores by 0.0661, 0.0751, and 0.0718, respectively. Figure 11a shows that the F1 score does not fall below 0.9168 in any of the considered scenarios. When $N = 2$, the F1 score ranges between 0.983 and 0.993 with no other applications running in the background; in case of background noise, the F1 score varies between 0.946 and 0.987. Similarly, we measured F1 values of 0.977, 0.967, 0.973 for $N \in \{3, 4, 5\}$ and observed that it falls to 0.917 in the worst case for $N = 4$ while running `x265 video encoder` in the background.

Figure 11b shows results with a larger window ($w = 256$). As discussed above, increasing w impacts detection latency; however, the performance gain justifies the choice of a larger observation window in most cases. When no applications are running in the background, the F1 score ranges between 0.989 and 0.993 for $N = 2$ and reaches 0.995, 0.988, and 0.991 for $N \in \{3, 4, 5\}$, respectively. In scenarios where applications are running in the background, F1 drops as low as 0.931 for $N = 2$ and $m = 8$ when the clones are running along `OpenCV`.

Figure 11 shows results up to $N = 5$, since the number of enclaves using CloneBuster on a given host at a given time is bounded by the numbers of available cores. In particular, this limitation comes from the registers used by the counting thread. Each core has only one such register, and it is shared by its two hyper-threads. As a consequence, as soon as the number of running enclaves using CloneBuster is greater than the number of physical cores, multiple context switches are required. The result is an increase in the number of asynchronous exits or a counter that does not increase between consecutive reads. Both cases should be labeled as an anomaly and appropriate countermeasure should be taken.

8 CONCLUSION

In this work, we addressed the problem of forking attacks against Intel SGX by cloning the victim enclave. We have analyzed 72 SGX-based applications and found that roughly 20% are vulnerable to such attacks, including those that rely on monotonic counters to prevent forking attacks based on rollbacks. A comprehensive solution to forking attacks requires a trusted third party that, unfortunately, are hard to find in real-world deployments.

To address this problem, we introduced CloneBuster, the first practical clone detection mechanism for SGX enclaves that does not rely on a trusted third party. As such, CloneBuster can be deployed immediately in any application scenario. We analyzed the security of CloneBuster and showed that a malicious OS cannot bypass it to spawn clones without detection. We implemented CloneBuster and evaluated its performance in existing SGX applications and under various realistic workloads. Our evaluation results show that CloneBuster achieves high accuracy in detecting clones, only incurs a marginal performance overheard, and adds up to 800 LoC to the TCB.

REFERENCES

[1] 2015. SpeicherDPDK. https://github.com/mbailleu/SpeicherDPDK.

[2] 2016. Luckychain. https://github.com/luckychain/lucky.

[3] 2016. SafeBricks. https://github.com/YangZhou1997/SafeBricks.

[4] 2016. Town Crier: An Authenticated Data Feed For Smart Contracts. https://github.com/bl4ck5un/Town-Crier.

[5] 2016. TresorSGX. https://github.com/ayeks/TresorSGX.

[6] 2017. Ekiden. https://github.com/ekiden/ekiden.

[7] 2017. NeXUS. https://github.com/sporgj/nexus-code.

[8] 2017. ObliDB. https://github.com/SabaEskandarian/ObliDB.

[9] 2017. Obscuro. https://github.com/BitObscuro/Obscuro.

[10] 2017. Opaque. https://github.com/mc2-project/opaque-sql.

[11] 2017. Private Contact Discovery Service (Beta). https://github.com/signalapp/ContactDiscoveryService.

[12] 2017. SGX Enabled OpenStack Barbican Key Management System. https://github.com/cloud-security-research/sgx-kms.

[13] 2017. sgx-papers. https://github.com/vschiavoni/sgx-papers.

[14] 2017. SGX-Tor. https://github.com/kaist-ina/SGX-Tor.

[15] 2017. StealthDB. https://github.com/cryptograph/stealthdb.

[16] 2018. BI-SGX : Bioinformatic Interpreter on SGX-based Secure Computing Cloud. https://github.com/hello31337/BI-SGX.

[17] 2018. Cloud Key Store - secure storage for private credentials. https://github.com/cloud-key-store/keystore.

[18] 2018. LightBox. https://github.com/lightbox-impl/LightBox.

[19] 2018. Oasis Core. https://github.com/oasisprotocol/oasis-core.

[20] 2018. POSUP: Oblivious Search and Update Platform with SGX. https://github.com/thanghoang/POSUP.

[21] 2018. SafeKeeper - Protecting Web passwords using Trusted Execution Environments. https://github.com/SafeKeeper/safekeeper-server.

[22] 2018. ShieldStore. https://github.com/cocoppang/ShieldStore.

[23] 2018. SkSES. https://github.com/ndokmai/sgx-genome-variants-search.

[24] 2019. Awesome SGX Open Source Projects. https://github.com/Maxul/Awesome-SGX-Open-Source.

[25] 2019. Boolean Isolated Searchable Encryption (BISEN). https://github.com/bernymac/BISEN.

[26] 2019. ConsenSGX. https://github.com/sshsshy/ConsenSGX.

[27] 2019. Phala Blockchain. https://github.com/Phala-Network/phala-blockchain.

[28] 2019. The SELIS Publish/Subscribe system. https://github.com/selisproject/pubsub.

[29] 2020. Plinius. https://github.com/anonymous-xh/plinius.

[30] 2020. QShield. https://github.com/fishermano/QShield.

[31] 2020. Secure XGBoost. https://github.com/mc2-project/secure-xgboost.

[32] 2020. SENG, the SGX-Enforcing Network Gateway. https://github.com/sengsgx/sengsgx.

[33] 2020. SGXSSE Maiden. https://github.com/MonashCybersecurityLab/SGXSSE.

[34] 2020. SMac: Secure Genotype Imputation in Intel SGX. https://github.com/ndokmai/sgx-genotype-imputation.

[35] 2020. STANlite. https://github.com/ibr-ds/STANlite.

[36] 2020. X-Search. https://github.com/Sand-jrd/SGX-Search.

[37] 2021. Accelerating Encrypted Deduplication via SGX. https://github.com/jingwei87/sgxdedup.

[38] 2021. Avocado. https://github.com/mbailleu/avocado.

[39] 2021. bwa-sgx-scone. https://github.com/dsc-sgx/bwa-sgx-scone.

[40] 2021. Desearch. https://github.com/SJTU-IPADS/DeSearch.

[41] 2021. Mechanics of MobileCoin: First Edition. https://mobilecoin.com/learn/read-the-whitepapers/mechanics/. Accessed: 23-02-2023.

[42] 2021. Snoopy: A Scalable Oblivious Storage System. https://github.com/ucbrise/snoopy.

[43] 2022. Artifact for paper #1520 SOTER: Guarding Black-box Inference for General Neural Networks at the Edge. https://github.com/hku-systems/SOTER.

[44] 2022. DEBE. https://github.com/yzr95924/DEBE.

[45] 2022. FeIDo Credential Service, Intel SGX version. https://github.com/feido-token.

[46] 2022. Implementation of the paper "Differentially-Private Payment Channels with Twilight". https://github.com/saart/Twilight.

[47] 2022. REX: SGX decentralized recommender. https://github.com/rafaelppires/rex.

[48] 2022. V2V SGX. https://github.com/OSUSecLab/v2v-sgx-prelim.

[49] 2023. CACIC Use Case. https://github.com/GTA-UFRJ/CACIC-Use-Case.

[50] 2023. No Forking Way: Detecting Cloning Attacks on Intel SGX Applications - Full Version. https://anonymous.4open.science/r/Clonebuster-F2F7/.

[51] Andreas Abel and Jan Reineke. 2019. uops.info: Characterizing Latency, Throughput, and Port Usage of Instructions on Intel Microarchitectures. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (Providence, RI, USA) *(ASPLOS '19)*. ACM, New York, NY, USA, 673–686. https://doi.org/10.1145/3297858.3304062

[52] Fritz Alder, N. Asokan, Arseny Kurnikov, Andrew Paverd, and Michael Steiner. 2019. S-FaaS: Trustworthy and Accountable Function-as-a-Service using Intel SGX. In *Proceedings of the 2019 ACM SIGSAC Conference on Cloud Computing Security Workshop, CCSW@CCS 2019, London, UK, November 11, 2019*. ACM, 185–199. https://doi.org/10.1145/3338466.3358916

[53] Julien Amacher and Valerio Schiavoni. 2019. On the Performance of ARM TrustZone - (Practical Experience Report). In *IFIP WG 6.1 International Conference on Distributed Applications and Interoperable Systems (DAIS)*. 133–151.

[54] Sergei Arnautov, Andrey Brito, Pascal Felber, Christof Fetzer, Franz Gregor, Robert Krahn, Wojciech Ozga, André Martin, Valerio Schiavoni, Fábio Silva, Marcus Tenorio, and Nikolaus Thümmel. 2018. PubSub-SGX: Exploiting Trusted Execution Environments for Privacy-Preserving Publish/Subscribe Systems. In *2018 IEEE 37th Symposium on Reliable Distributed Systems (SRDS)*. 123–132. https://doi.org/10.1109/SRDS.2018.00023

[55] Maurice Bailleu, Dimitra Giantsidi, Vasilis Gavrielatos, Do Le Quoc, Vijay Nagarajan, and Pramod Bhatotia. 2021. Avocado: A Secure In-Memory Distributed Storage System.. In *USENIX Annual Technical Conference*. 65–79.

[56] Maurice Bailleu, Jörg Thalheim, Pramod Bhatotia, Christof Fetzer, Michio Honda, and Kapil Vaswani. 2019. SPEICHER: Securing LSM-based Key-Value Stores using Shielded Execution.. In *FAST*. 173–190.

[57] Johannes Behl, Tobias Distler, and Rüdiger Kapitza. 2017. Hybrids on Steroids: SGX-Based High Performance BFT. In *Proceedings of the Twelfth European Conference on Computer Systems* (Belgrade, Serbia) *(EuroSys '17)*. Association for Computing Machinery, New York, NY, USA, 222–237. https://doi.org/10.1145/3064176.3064213

[58] Andrea Bittau, Úlfar Erlingsson, Petros Maniatis, Ilya Mironov, Ananth Raghunathan, David Lie, Mitch Rudominer, Ushasree Kode, Julien Tinnes, and Bernhard Seefeld. 2017. Prochlo: Strong Privacy for Analytics in the Crowd. In *Proceedings of the 26th Symposium on Operating Systems Principles* (Shanghai, China) *(SOSP '17)*. Association for Computing Machinery, New York, NY, USA, 441–459. https://doi.org/10.1145/3132747.3132769

[59] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. 2018. Private data objects: an overview. *arXiv preprint arXiv:1807.05686* (2018).

[60] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. 2018. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *arXiv preprint arXiv:1805.08541* (2018).

[61] Marcus Brandenburger, Christian Cachin, Matthias Lorenz, and Rüdiger Kapitza. 2017. Rollback and Forking Detection for Trusted Execution Environments Using Lightweight Collective Memory. In *47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 157–168.

[62] Samira Briongos, Pedro Malagon, Jose M. Moya, and Thomas Eisenbarth. 2020. RELOAD+REFRESH: Abusing Cache Replacement Policies to Perform Stealthy Cache Attacks. In *29th USENIX Security Symposium (USENIX Security 20)*. USENIX Association, Boston, MA. https://www.usenix.org/conference/usenixsecurity20/presentation/briongos

[63] David Cerdeira, Nuno Santos, Pedro Fonseca, and Sandro Pinto. 2020. SoK: Understanding the Prevailing Security Vulnerabilities in TrustZone-assisted TEE Systems. In *2020 IEEE Symposium on Security and Privacy (SP)*. 1416–1432. https://doi.org/10.1109/SP40000.2020.00061

[64] Somnath Chakrabarti, Brandon Baker, and Mona Vij. 2017. Intel SGX enabled key manager service with openstack barbican. *arXiv preprint arXiv:1712.07694* (2017).

[65] G. Chen, S. Chen, Y. Xiao, Y. Zhang, Z. Lin, and T. H. Lai. 2019. SgxPectre: Stealing Intel Secrets from SGX Enclaves Via Speculative Execution. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 142–157. https://doi.org/10.1109/EuroSP.2019.00020

[66] Ju Chen, Yuzhe Richard Tang, and Hao Zhou. 2017. Strongly Secure and Efficient Data Shuffle on Hardware Enclaves. In *2nd Workshop on System Software for Trusted Execution (SysTEX)*. 1:1–1:6.

[67] Lixia Chen, Jian Li, Ruhui Ma, Haibing Guan, and Hans-Arno Jacobsen. 2019. EnclaveCache: A Secure and Scalable Key-Value Cache in Multi-Tenant Clouds Using Intel SGX *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 14–27. https://doi.org/10.1145/3361525.3361533

[68] Sanchuan Chen, Fangfei Liu, Zeyu Mi, Yinqian Zhang, Ruby B. Lee, Haibo Chen, and XiaoFeng Wang. 2018. Leveraging Hardware Transactional Memory for Cache Side-Channel Defenses. In *Asia Conference on Computer and Communications Security (AsiaCCS)*. 601–608.

[69] Sanchuan Chen, Xiaokuan Zhang, Michael K. Reiter, and Yinqian Zhang. 2017. Detecting Privileged Side-Channel Attacks in Shielded Execution with DéJà Vu *(ASIA CCS '17)*. Association for Computing Machinery, New York, NY, USA, 7–18. https://doi.org/10.1145/3052973.3053007

[70] Yaxing Chen, Qinghua Zheng, Zheng Yan, and Dan Liu. 2021. QShield: Protecting Outsourced Cloud Data Queries With Multi-User Access Control Based on SGX. *IEEE Transactions on Parallel and Distributed Systems* 32, 2 (2021), 485–499. https://doi.org/10.1109/TPDS.2020.3024880

[71] Raymond Cheng, Fan Zhang, Jernej Kos, Warren He, Nicholas Hynes, Noah Johnson, Ari Juels, Andrew Miller, and Dawn Song. 2019. Ekiden: A Platform for Confidentiality-Preserving, Trustworthy, and Performant Smart Contracts.

In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*. 185–200. https://doi.org/10.1109/EuroSP.2019.00023

[72] Haehyun Cho, Penghui Zhang, Donguk Kim, Jinbum Park, Choong-Hoon Lee, Ziming Zhao, Adam Doupé, and Gail-Joon Ahn. 2018. Prime+Count: Novel Cross-World Covert Channels on ARM TrustZone. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 441–452. https://doi.org/10.1145/3274694.3274704

[73] Simon Da Silva, Sonia Ben Mokhtar, Stefan Contiu, Daniel Négru, Laurent Réveillère, and Etienne Rivière. 2019. PrivaTube: Privacy-Preserving Edge-Assisted Video Streaming. In *Proceedings of the 20th International Middleware Conference* (Davis, CA, USA) *(Middleware '19)*. Association for Computing Machinery, New York, NY, USA, 189–201. https://doi.org/10.1145/3361525.3361546

[74] Emma Dauterman, Vivian Fang, Ioannis Demertzis, Natacha Crooks, and Raluca Ada Popa. 2021. Snoopy: Surpassing the Scalability Bottleneck of Oblivious Storage. In *Proceedings of the ACM SIGOPS 28th Symposium on Operating Systems Principles* (Virtual Event, Germany) *(SOSP '21)*. Association for Computing Machinery, New York, NY, USA, 655–671. https://doi.org/10.1145/3477132.3483562

[75] Akash Dhasade, Nevena Dresevic, Anne-Marie Kermarrec, and Rafael Pires. 2022. TEE-based decentralized recommender systems: The raw data sharing redemption. In *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. 447–458. https://doi.org/10.1109/IPDPS53621.2022.00050

[76] Judicael B. Djoko, Jack Lange, and Adam J. Lee. 2019. NeXUS: Practical and Secure Access Control on Untrusted Storage Platforms using Client-Side SGX. In *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 401–413. https://doi.org/10.1109/DSN.2019.00049

[77] AMD (Tech Docs). 2018. SEV Secure Nested Paging Firmware ABI Specification. https://www.amd.com/system/files/TechDocs/SEV-SNP-strengthening-vm-isolation-with-integrity-protection-and-more.pdf. Accessed: 2022-10-01.

[78] AMD (Tech Docs). 2021. AMD64 Architecture Programmer's Manual Volume 2: System Programming. https://www.amd.com/system/files/TechDocs/24593.pdf.

[79] Natnatee Dokmai, Can Kockan, Kaiyuan Zhu, XiaoFeng Wang, S Cenk Sahinalp, and Hyunghoon Cho. 2021. Privacy-preserving genotype imputation in a trusted execution environment. *Cell systems* 12, 10 (2021), 983–993.

[80] Maya Dotan, Saar Tochner, Aviv Zohar, and Yossi Gilad. 2022. Twilight: A Differentially Private Payment Channel Network. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 555–570. https://www.usenix.org/conference/usenixsecurity22/presentation/dotan

[81] Huayi Duan, Cong Wang, Xingliang Yuan, Yajin Zhou, Qian Wang, and Kui Ren. 2019. LightBox: Full-Stack Protected Stateful Middlebox at Lightning Speed. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security* (London, United Kingdom) *(CCS '19)*. Association for Computing Machinery, New York, NY, USA, 2351–2367. https://doi.org/10.1145/3319535.3339814

[82] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 502–518.

[83] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).

[84] Bernardo Ferreira, Bernardo Portela, Tiago Oliveira, Guilherme Borges, Henrique Domingos, and João Leitão. 2022. Boolean Searchable Symmetric Encryption With Filters on Trusted Hardware. *IEEE Transactions on Dependable and Secure Computing* 19, 2 (2022), 1307–1319. https://doi.org/10.1109/TDSC.2020.3012100

[85] MobileCoin Foundation. 2019. MobileCoin. https://github.com/mobilecoinfoundation/mobilecoin.

[86] Benny Fuhry, Raad Bahmani, Ferdinand Brasser, Florian Hahn, Florian Kerschbaum, and Ahmad-Reza Sadeghi. 2017. HardIDX: Practical and Secure Index with SGX. In *Data and Applications Security and Privacy XXXI*. Springer International Publishing, Cham, 386–408.

[87] Benny Fuhry, Lina Hirschoff, Samuel Koesnadi, and Florian Kerschbaum. 2020. SeGShare: Secure Group File Sharing in the Cloud using Enclaves. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 476–488. https://doi.org/10.1109/DSN48063.2020.00061

[88] Qian Ge, Yuval Yarom, David Cock, and Gernot Heiser. 2018. A survey of microarchitectural timing attacks and countermeasures on contemporary hardware. *J. Cryptogr. Eng.* 8, 1 (2018), 1–27. https://doi.org/10.1007/s13389-016-0141-6

[89] giganetom et al. 2018. SATisPy. https://github.com/netom/satispy. Accessed: 2020-10-01.

[90] David Goltzsche, Signe Rüsch, Manuel Nieke, Sébastien Vaucher, Nico Weichbrodt, Valerio Schiavoni, Pierre-Louis Aublin, Paolo Cosa, Christof Fetzer, Pascal Felber, Peter Pietzuch, and Rüdiger Kapitza. 2018. EndBox: Scalable Middlebox Functions Using Client-Side Trusted Execution. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 386–397. https://doi.org/10.1109/DSN.2018.00048

[91] Daniel Gruss, Julian Lettner, Felix Schuster, Olya Ohrimenko, Istvan Haller, and Manuel Costa. 2017. Strong and Efficient Cache Side-Channel Protection using Hardware Transactional Memory. In *26th USENIX Security Symposium (USENIX Security 17)*. USENIX Association, Vancouver, BC, 217–233. https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/gruss

[92] Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2016. Rowhammer.js: A Remote Software-Induced Fault Attack in JavaScript. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 300–321.

[93] Daniel Hackenberg, Robert Schöne, Thomas Ilsche, Daniel Molka, Joseph Schuchart, and Robin Geyer. 2015. An Energy Efficiency Feature Survey of the Intel Haswell Processor. In *2015 IEEE International Parallel and Distributed Processing Symposium Workshop*. 896–904. https://doi.org/10.1109/IPDPSW.2015.70

[94] Andrew Herdrich, Edwin Verplanke, Priya Autee, Ramesh Illikkal, Chris Gianos, Ronak Singhal, and Ravi Iyer. 2016. Cache QoS: From concept to reality in the Intel® Xeon® processor E5-2600 v3 product family. In *2016 IEEE International Symposium on High Performance Computer Architecture (HPCA)*. 657–668. https://doi.org/10.1109/HPCA.2016.7446102

[95] Thang Hoang, Muslum Ozgur Ozmen, Yeongjin Jang, and Attila A Yavuz. 2019. Hardware-supported ORAM in effect: Practical oblivious search and update on very large dataset. *Proceedings on Privacy Enhancing Technologies* 2019, 1 (2019), 172–191.

[96] Casen Hunger, Mikhail Kazdagli, Ankit Rawat, Alex Dimakis, Sriram Vishwanath, and Mohit Tiwari. 2015. Understanding contention-based channels and using them for defense. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*. 639–650. https://doi.org/10.1109/HPCA.2015.7056069

[97] Hyperledger. 2018. Hyperledger Fabric Private Chaincode. https://github.com/hyperledger/fabric-private-chaincode.

[98] Intel(R). 2015. Improving Real-Time Performance by Utilizing Cache Allocation Technology. https://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf. Accessed March 22.

[99] Intel(R). 2018. Intel(R)64 and IA-32 Architectures Optimization Reference Manual. https://software.intel.com/content/www/us/en/develop/download/intel-64-and-ia-32-architectures-optimization-reference-manual.html. Accessed: 2020-10-01.

[100] Intel(R). 2021. Intel(R) RDT Software Package. https://github.com/intel/intel-cmt-cat.

[101] Intel(R). 2021. Intel® Xeon® Scalable Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html.

[102] Intel(R). 2022. Intel® Product Specifications. https://ark.intel.com/content/www/us/en/ark/search/featurefilter.html?productType=873&2_SoftwareGuardExtensions=Yes%20with%20Intel%C2%AE%20SPS&1_Filter-Family=595

[103] Gorka Irazoqui, Thomas Eisenbarth, and Berk Sunar. 2015. S$A: A Shared Cache Attack That Works across Cores and Defies VM Sandboxing – and Its Application to AES. In *2015 IEEE Symposium on Security and Privacy*. 591–604. https://doi.org/10.1109/SP.2015.42

[104] G. Irazoqui, T. Eisenbarth, and B. Sunar. 2015. Systematic Reverse Engineering of Cache Slice Selection in Intel Processors. In *2015 Euromicro Conference on Digital System Design*. 629–636.

[105] Saad Islam, Ahmad Moghimi, Ida Bruhns, Moritz Krebbel, Berk Gulmezoglu, Thomas Eisenbarth, and Berk Sunar. 2019. SPOILER: Speculative Load Hazards Boost Rowhammer and Cache Attacks. In *28th USENIX Security Symposium (USENIX Security 19)*. USENIX Association, Santa Clara, CA, 621–637. https://www.usenix.org/conference/usenixsecurity19/presentation/islam

[106] M Jangid and Zhiqiang Lin. 2022. Towards a TEE-based V2V Protocol for Connected and Autonomous Vehicles. In *Workshop on Automotive and Autonomous Vehicle Security (AutoSec)*.

[107] Mohit Kumar Jangid, Guoxing Chen, Yinqian Zhang, and Zhiqiang Lin. 2021. Towards Formal Verification of State Continuity for Enclave Programs. In *30th USENIX Security Symposium (USENIX Security 21)*. 573–590.

[108] Prasad Koshy Jose. 2020. Confidential Computing of Machine Learning using Intel SGX. https://github.com/prasadkjose/confidential-ml-sgx.

[109] Gabriel Kaptchuk, Matthew Green, and Ian Miers. 2019. Giving State to the Stateless: Augmenting Trustworthy Computation with Ledgers. In *Network and Distributed System Security Symposium, (NDSS)*. 1–15.

[110] Seongmin Kim, Juhyeng Han, Jaehyeong Ha, Taesoo Kim, and Dongsu Han. 2018. SGX-Tor: A Secure and Practical Tor Anonymity Network With SGX Enclaves. *IEEE/ACM Transactions on Networking* 26, 5 (2018), 2174–2187. https://doi.org/10.1109/TNET.2018.2868054

[111] Taehoon Kim, Joongun Park, Jaewook Woo, Seungheun Jeon, and Jaehyuk Huh. 2019. ShieldStore: Shielded In-Memory Key-Value Storage with SGX. In *Proceedings of the Fourteenth EuroSys Conference 2019* (Dresden, Germany) *(EuroSys '19)*. Association for Computing Machinery, New York, NY, USA, Article 14, 15 pages. https://doi.org/10.1145/3302424.3303951

[112] Felix Kirchengast. 2019. Secure Network Interface with SGX. https://github.com/fkirc/secure-network-interface-with-sgx. *GitHub repository* (2019).

[113] Can Kockan, Kaiyuan Zhu, Natnatee Dokmai, Nikolai Karpov, M Oguzhan Kulekci, David P Woodruff, and S Cenk Sahinalp. 2020. Sketching algorithms for genomic data analysis and querying in a secure enclave. *Nature methods* 17, 3 (2020), 295–301.

[114] Robert Krahn, Bohdan Trach, Anjo Vahldiek-Oberwagner, Thomas Knauth, Pramod Bhatotia, and Christof Fetzer. 2018. Pesos: Policy Enhanced Secure Object Store. In *Proceedings of the Thirteenth EuroSys Conference* (Porto, Portugal) *(EuroSys '18)*. Association for Computing Machinery, New York, NY, USA, Article 25, 17 pages. https://doi.org/10.1145/3190508.3190518

[115] Klaudia Krawiecka, Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. SafeKeeper: Protecting Web Passwords Using Trusted Execution Environments. In *Proceedings of the 2018 World Wide Web Conference* (Lyon, France) *(WWW '18)*. International World Wide Web Conferences Steering Committee, Republic and Canton of Geneva, CHE, 349–358. https://doi.org/10.1145/3178876.3186101

[116] Arseny Kurnikov, Andrew Paverd, Mohammad Mannan, and N. Asokan. 2018. Keys in the Clouds: Auditable Multi-Device Access to Cryptographic Credentials. In *Proceedings of the 13th International Conference on Availability, Reliability and Security* (Hamburg, Germany) *(ARES 2018)*. Association for Computing Machinery, New York, NY, USA, Article 40, 10 pages. https://doi.org/10.1145/3230833.3234518

[117] Donghyun Kwon, Jiwon Seo, Yeongpil Cho, Byoungyoung Lee, and Yunheung Paek. 2020. PrOS: Light-Weight Privatized Se cure OSes in ARM TrustZone. *IEEE Trans. Mob. Comput.* 19, 6 (2020), 1434–1447.

[118] Hyperledger Labs. 2018. Hyperledger Private Data Objects. https://github.com/hyperledger-labs/private-data-objects.

[119] Fan Lang, Wei Wang, Lingjia Meng, Jingqiang Lin, Qiongxiao Wang, and Linli Lu. 2022. MoLE: Mitigation of Side-channel Attacks against SGX via Dynamic Data Location Escape. In *Annual Computer Security Applications Conference (ACSAC)*. 978–988.

[120] Michael Larabel and Matthew Tippett. 2008. Phoronix Test Suite. http://www.phoronix-test-suite.com/.

[121] Andrew Law, Chester Leung, Rishabh Poddar, Raluca Ada Popa, Chenyu Shi, Octavian Sima, Chaofan Yu, Xingmeng Zhang, and Wenting Zheng. 2020. Secure Collaborative Training and Inference for XGBoost. In *Proceedings of the 2020 Workshop on Privacy-Preserving Machine Learning in Practice* (Virtual Event, USA) *(PPMLP'20)*. Association for Computing Machinery, New York, NY, USA, 21–26. https://doi.org/10.1145/3411501.3419420

[122] Mingyu Li, Jinhao Zhu, Tianxu Zhang, Cheng Tan, Yubin Xia, Sebastian Angel, and Haibo Chen. 2021. Bringing Decentralized Search to Decentralized Services.. In *OSDI*. 331–347.

[123] Joshua Lind. 2018. Teechain: A Secure Payment Network with Asynchronous Blockchain Access. https://github.com/lsds/Teechain.

[124] Joshua Lind, Ittay Eyal, Florian Kelbert, Oded Naor, Peter Pietzuch, and Emin Gün Sirer. 2017. Teechain: Scalable blockchain payments using trusted execution environments. *arXiv preprint arXiv:1707.05454* (2017).

[125] Joshua Lind, Oded Naor, Ittay Eyal, Florian Kelbert, Emin Gün Sirer, and Peter Pietzuch. 2019. Teechain: A Secure Payment Network with Asynchronous Blockchain Access *(SOSP '19)*. Association for Computing Machinery, New York, NY, USA, 63–79. https://doi.org/10.1145/3341301.3359627

[126] Moritz Lipp, Daniel Gruss, Raphael Spreitzer, Clémentine Maurice, and Stefan Mangard. 2016. ARMageddon: Cache Attacks on Mobile Devices. In *25th USENIX Security Symposium (USENIX Security 16)*. USENIX Association, Austin, TX, 549–564. https://www.usenix.org/conference/usenixsecurity16/technical-sessions/presentation/lipp

[127] Moritz Lipp, Vedad Hadžić, Michael Schwarz, Arthur Perais, Clémentine Maurice, and Daniel Gruss. 2020. Take A Way: Exploring the Security Implications of AMD's Cache Way Predictors. In *Proceedings of the 15th ACM Asia Conference on Computer and Communications Security* (Taipei, Taiwan) *(ASIA CCS '20)*. Association for Computing Machinery, New York, NY, USA, 813–825. https://doi.org/10.1145/3320269.3384746

[128] Fangfei Liu, Yuval Yarom, Qian Ge, Gernot Heiser, and Ruby B. Lee. 2015. Last-Level Cache Side-Channel Attacks Are Practical. In *Proceedings of the 2015 IEEE Symposium on Security and Privacy (SP '15)*. IEEE Computer Society, USA, 605–622. https://doi.org/10.1109/SP.2015.43

[129] Rudolf Loretan. 2021. Enclave hardening for private ML. https://github.com/loretanr/dp-gbdt.

[130] Moxie Marlinspike. 2017. Technology preview: Private contact discovery for Signal. https://signal.org/blog/private-contact-discovery/. (2017). Accessed: 09-03-2023.

[131] Sinisa Matetic, Mansoor Ahmed, Kari Kostiainen, Aritra Dhar, David Sommer, Arthur Gervais, Ari Juels, and Srdjan Capkun. 2017. ROTE: Rollback Protection for Trusted Execution. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 1289–1306.

[132] Sinisa Matetic, Moritz Schneider, Andrew Miller, Ari Juels, and Srdjan Capkun. 2018. DelegaTEE: Brokered Delegation Using Trusted Execution Environments.. In *USENIX Security Symposium*. 1387–1403.

[133] Sinisa Matetic, Karl Wüst, Moritz Schneider, Kari Kostiainen, Ghassan Karame, and Srdjan Capkun. 2019. BITE: Bitcoin Lightweight Client Privacy using Trusted Execution.. In *USENIX Security Symposium*. 783–800.

[134] Clémentine Maurice, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. C5: Cross-Cores Cache Covert Channel. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, Cham, 46–64.

[135] Clémentine Maurice, Nicolas Le Scouarnec, Christoph Neumann, Olivier Heen, and Aurélien Francillon. 2015. Reverse Engineering Intel Last-Level Cache Complex Addressing Using Performance Counters. In *Research in Attacks, Intrusions, and Defenses - 18th International Symposium, RAID 2015, Kyoto, Japan, November 2-4, 2015, Proceedings*. 48–65.

[136] Mitar Milutinovic, Warren He, Howard Wu, and Maxinder Kanwal. 2016. Proof of Luck: An Efficient Blockchain Consensus Protocol. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (Trento, Italy) *(SysTEX '16)*. Association for Computing Machinery, New York, NY, USA, Article 2, 6 pages. https://doi.org/10.1145/3007788.3007790

[137] Ahmad Moghimi, Gorka Irazoqui, and Thomas Eisenbarth. 2017. CacheZoom: How SGX Amplifies the Power of Cache Attacks. In *Cryptographic Hardware and Embedded Systems – CHES 2017*. Springer International Publishing, Cham, 69–90.

[138] Sonia Ben Mokhtar, Antoine Boutet, Pascal Felber, Marcelo Pasin, Rafael Pires, and Valerio Schiavoni. 2017. X-Search: Revisiting Private Web Search Using Intel SGX. In *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference* (Las Vegas, Nevada) *(Middleware '17)*. Association for Computing Machinery, New York, NY, USA, 198–208. https://doi.org/10.1145/3135974.3135987

[139] Jianyu Niu, Wei Peng, Xiaokuan Zhang, and Yinqian Zhang. 2022. NARRATOR: Secure and Practical State Continuity for Trusted Execution in the Cloud. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security, CCS 2022, Los Angeles, CA, USA, November 7-11, 2022*. 2385–2399.

[140] Yossef Oren, Vasileios P. Kemerlis, Simha Sethumadhavan, and Angelos D. Keromytis. 2015. The Spy in the Sandbox: Practical Cache Attacks in JavaScript and Their Implications. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security* (Denver, Colorado, USA) *(CCS '15)*. Association for Computing Machinery, New York, NY, USA, 1406–1418. https://doi.org/10.1145/2810103.2813708

[141] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay. 2011. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research* 12 (2011), 2825–2830.

[142] Peter Pessl, Daniel Gruss, Clémentine Maurice, Michael Schwarz, and Stefan Mangard. 2016. DRAMA: Exploiting Dram Addressing for Cross-Cpu Attacks. In *Proceedings of the 25th USENIX Conference on Security Symposium* (Austin, TX, USA) *(SEC'16)*. USENIX Association, USA, 565–581.

[143] Sandro Pinto and Nuno Santos. 2019. Demystifying Arm TrustZone: A Comprehensive Survey. 51, 6, Article 130 (jan 2019), 36 pages. https://doi.org/10.1145/3291047

[144] Rafael Pires, David Goltzsche, Sonia Ben Mokhtar, Sara Bouchenak, Antoine Boutet, Pascal Felber, Rüdiger Kapitza, Marcelo Pasin, and Valerio Schiavoni. 2018. CYCLOSA: Decentralizing Private Web Search through SGX-Based Browser Extensions. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. 467–477. https://doi.org/10.1109/ICDCS.2018.00053

[145] Rafael Pires, Marcelo Pasin, Pascal Felber, and Christof Fetzer. 2016. Secure Content-Based Routing Using Intel Software Guard Extensions. In *Proceedings of the 17th International Middleware Conference* (Trento, Italy) *(Middleware '16)*. Association for Computing Machinery, New York, NY, USA, Article 10, 10 pages. https://doi.org/10.1145/2988336.2988346

[146] Rishabh Poddar, Chang Lan, Raluca Ada Popa, and Sylvia Ratnasamy. 2018. Safebricks: Shielding network functions in the cloud. In *15th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 18)*. 201–216.

[147] C. Priebe, K. Vaswani, and M. Costa. 2018. EnclaveDB: A Secure Database Using SGX. In *2018 IEEE Symposium on Security and Privacy (SP)*. 264–278. https://doi.org/10.1109/SP.2018.00025

[148] Pengfei Qiu, Dongsheng Wang, Yongqiang Lyu, Ruidong Tian, Chunlu Wang, and Gang Qu. 2021. VoltJockey: A New Dynamic Voltage Scaling-Based Fault Injection Attack on Intel SGX. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 40, 6 (2021), 1130–1143. https://doi.org/10.1109/TCAD.2020.3024853

[149] Do Le Quoc, Franz Gregor, Sergei Arnautov, Roland Kunkel, Pramod Bhatotia, and Christof Fetzer. 2020. SecureTF: A Secure TensorFlow Framework. In *Proceedings of the 21st International Middleware Conference* (Delft, Netherlands) *(Middleware '20)*. Association for Computing Machinery, New York, NY, USA, 44–59. https://doi.org/10.1145/3423211.3425687

[150] Yanjing Ren, Jingwei Li, Zuoru Yang, Patrick PC Lee, and Xiaosong Zhang. 2021. SGXDedup. In *USENIX Annual Technical Conference*. 957–971.

[151] Lars Richter, Johannes Götzfried, and Tilo Müller. 2016. Isolating Operating System Components with Intel SGX. In *Proceedings of the 1st Workshop on System Software for Trusted Execution* (Trento, Italy) *(SysTEX '16)*. Association for Computing Machinery, New York, NY, USA, Article 8, 6 pages. https://doi.org/10.1145/3007788.3007796

[152] Signe Rüsch, Kai Bleeke, and Rüdiger Kapitza. 2019. Bloxy: Providing Transparent and Generic BFT-Based Ordering Services for Blockchains. In *2019 38th Symposium on Reliable Distributed Systems (SRDS)*. 305–30509. https://doi.org/10.1109/SRDS47363.2019.00043

[153] Aoi Sakurai. 2019. BI-SGX: Secure Cloud Computation. https://github.com/hello31337/BI-SGX. Accessed: 2023-01-16.

[154] Vasily Sartakov, Nico Weichbrodt, Sebastian Krieter, Thomas Leich, and Rudiger Kapitza. 2018. STANlite – A Database Engine for Secure Data Processing at Rack-Scale Level. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*. 23–33. https://doi.org/10.1109/IC2E.2018.00024

[155] Sajin Sasy and Ian Goldberg. 2019. ConsenSGX: Scaling Anonymous Communications Networks with Trusted Execution Environments. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 331–349.

[156] Sajin Sasy, Aaron Johnson, and Ian Goldberg. 2022. Fast Fully Oblivious Compaction and Shuffling. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2565–2579. https://doi.org/10.1145/3548606.3560603

[157] Robert Schone, Thomas Ilsche, Mario Bielert, Andreas Gocht, and Daniel Hackenberg. 2019. Energy Efficiency Features of the Intel Skylake-SP Processor and Their Impact on Performance. *2019 International Conference on High Performance Computing & Simulation (HPCS)* (Jul 2019). https://doi.org/10.1109/hpcs48598.2019.9188239

[158] Fabian Schwarz, Khue Do, Gunnar Heide, Lucjan Hanzlik, and Christian Rossow. 2022. FeIDo: Recoverable FIDO2 Tokens Using Electronic IDs. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security* (Los Angeles, CA, USA) *(CCS '22)*. Association for Computing Machinery, New York, NY, USA, 2581–2594. https://doi.org/10.1145/3548606.3560584

[159] Fabian Schwarz and Christian Rossow. 2020. SENG, the sgx-enforcing network gateway: Authorizing communication from shielded clients. In *Proceedings of the 29th USENIX Conference on Security Symposium*. 753–770.

[160] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2017. Malware Guard Extension: Using SGX to Conceal Cache Attacks. In *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer International Publishing, Cham, 3–24.

[161] Michael Schwarz, Samuel Weiser, Daniel Gruss, Clémentine Maurice, and Stefan Mangard. 2020. Malware Guard Extension: abusing Intel SGX to conceal cache attacks. *Cybersecurity* 3, 2 (2020). https://doi.org/10.1186/s42400-019-0042-y

[162] Tianxiang Shen, Ji Qi, Jianyu Jiang, Xian Wang, Siyuan Wen, Xusheng Chen, Shixiong Zhao, Sen Wang, Li Chen, Xiapu Luo, Fengwei Zhang, and Heming Cui. 2022. SOTER: Guarding Black-box Inference for General Neural Networks at the Edge. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 723–738. https://www.usenix.org/conference/atc22/presentation/shen

[163] Ming-Wei Shih, Mohan Kumar, Taesoo Kim, and Ada Gavrilovska. 2016. S-NFV: Securing NFV States by Using SGX. In *Proceedings of the 2016 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization* (New Orleans, Louisiana, USA) *(SDN-NFV Security '16)*. Association for Computing Machinery, New York, NY, USA, 45–48. https://doi.org/10.1145/2876019.2876032

[164] Ming-Wei Shih, Sangho Lee, Taesoo Kim, and Marcus Peinado. 2017. T-SGX: Eradicating Controlled-Channel Attacks Against Enclave Programs. https://doi.org/10.14722/ndss.2017.23193

[165] C. Soriente, G. Karame, W. Li, and S. Fedorov. 2019. ReplicaTEE: Enabling Seamless Replication of SGX Enclaves in the Cloud. In *2019 IEEE European Symposium on Security and Privacy (EuroS P)*. 158–171. https://doi.org/10.1109/EuroSP.2019.00021

[166] Vasileios Spiliopoulos, Stefanos Kaxiras, and Georgios Keramidas. 2011. Green governors: A framework for Continuously Adaptive DVFS. In *2011 International Green Computing Conference and Workshops*. 1–8. https://doi.org/10.1109/IGCC.2011.6008552

[167] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium*. 875–892.

[168] Raoul Strackx and Frank Piessens. 2017. The Heisenberg Defense: Proactively Defending SGX Enclaves against Page-Table-Based Side-Channel Attacks. *CoRR* abs/1712.08519 (2017).

[169] Yuanyuan Sun, Sheng Wang, Huorong Li, and Feifei Li. 2021. Building Enclave-Native Storage Engines for Practical Encrypted Databases. *Proc. VLDB Endow.* 14, 6 (feb 2021), 1019–1032. https://doi.org/10.14778/3447689.3447705

[170] Guilherme A. Thomaz, Matheus B. Guerra, Matteo Sammarco, Marcin Detyniecki, and Miguel Elias M. Campista. 2022. Tamper-proof Access Control for IoT Clouds Using Enclaves. (2022). https://www.gta.ufrj.br/ftp/gta/TechReports/TGS23.pdf

[171] Bohdan Trach, Rasha Faqeh, Oleksii Oleksenko, Wojciech Ozga, Pramod Bhatotia, and Christof Fetzer. 2020. T-Lease: a trusted lease primitive for distributed systems. In *ACM Symposium on Cloud Computing (SoCC)*. 387–400.

[172] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).

[173] Florian Tramèr and Dan Boneh. 2018. SLALOM. https://github.com/ftramer/slalom.

[174] Muoi Tran, Loi Luu, Min Suk Kang, Iddo Bentov, and Prateek Saxena. 2018. Obscuro: A Bitcoin Mixer Using Trusted Execution Environments. In *Proceedings of the 34th Annual Computer Security Applications Conference* (San Juan, PR, USA) *(ACSAC '18)*. Association for Computing Machinery, New York, NY, USA, 692–701. https://doi.org/10.1145/3274694.3274750

[175] Jo Van Bulck, Frank Piessens, and Raoul Strackx. 2017. SGX-Step: A Practical Attack Framework for Precise Enclave Execution Control. In *Proceedings of the 2nd Workshop on System Software for Trusted Execution* (Shanghai, China) *(SysTEX'17)*. Association for Computing Machinery, New York, NY, USA, Article 4, 6 pages. https://doi.org/10.1145/3152701.3152706

[176] Jo Van Bulck, Nico Weichbrodt, Rüdiger Kapitza, Frank Piessens, and Raoul Strackx. 2017. Telling Your Secrets without Page Faults: Stealthy Page Table-Based Attacks on Enclaved Execution. In *Proceedings of the 26th USENIX Conference on Security Symposium* (Vancouver, BC, Canada) *(SEC'17)*. USENIX Association, USA, 1041–1056.

[177] Marten van Dijk, Jonathan Rhodes, Luis F. G. Sarmenta, and Srinivas Devadas. 2007. Offline untrusted storage with immediate detection of forking and replay attacks. In *ACM Workshop on Scalable Trusted Computing (STC)*. 41–48.

[178] Pepe Vila, Pierre Ganty, Marco Guarnieri, and Boris Köpf. 2019. CacheQuery: Learning Replacement Policies from Hardware Caches. *arXiv preprint arXiv:1912.09770* (2019).

[179] Pepe Vila, Boris Köpf, and José F. Morales. 2019. Theory and Practice of Finding Eviction Sets. In *2019 IEEE Symposium on Security and Privacy, SP 2019, San Francisco, CA, USA, May 19-23, 2019*. IEEE, 39–54. https://doi.org/10.1109/SP.2019.00042

[180] Dhinakaran Vinayagamurthy, Alexey Gribov, and Sergey Gorbunov. 2019. StealthDB: a Scalable Encrypted Database with Full SQL Query Support. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 370–388.

[181] Viet Vo, Shangqi Lai, Xingliang Yuan, Surya Nepal, and Joseph K. Liu. 2021. Towards Efficient and Strong Backward Private Searchable Encryption with Secure Enclaves. In *Applied Cryptography and Network Security*. Springer International Publishing, Cham, 50–75.

[182] Chathura Widanage, Weijie Liu, Jiayu Li, Hongbo Chen, XiaoFeng Wang, Haixu Tang, and Judy Fox. 2021. HySec-Flow: Privacy-Preserving Genomic Computing with SGX-based Big-Data Analytics Framework. In *2021 IEEE 14th International Conference on Cloud Computing (CLOUD)*. 733–743. https://doi.org/10.1109/CLOUD53861.2021.00098

[183] wolfSSL. 2017. wolfSSL Linux Enclave Example. https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux. Accessed: 2020-21-04.

[184] Zhenyu Wu, Zhang Xu, and Haining Wang. 2012. Whispers in the Hyper-space: High-speed Covert Channel Attacks in the Cloud. In *21st USENIX Security Symposium (USENIX Security 12)*. USENIX Association, Bellevue, WA, 159–173.

[185] Mengjia Yan, Read Sprabery, Bhargava Gopireddy, Christopher Fletcher, Roy Campbell, and Josep Torrellas. 2019. Attack Directories, Not Caches: Side Channel Attacks in a Non-Inclusive World. In *2019 IEEE Symposium on Security and Privacy (SP)*. 888–904. https://doi.org/10.1109/SP.2019.00004

[186] Fan Yang, Youmin Chen, Youyou Lu, Qing Wang, and Jiwu Shu. 2021. Aria: Tolerating Skewed Workloads in Secure In-memory Key-value Stores. In *37th IEEE International Conference on Data Engineering (ICDE)*. 1020–1031.

[187] Zuoru Yang, Jingwei Li, and Patrick P. C. Lee. 2022. Secure and Lightweight Deduplicated Storage via Shielded Deduplication-Before-Encryption. In *2022 USENIX Annual Technical Conference (USENIX ATC 22)*. USENIX Association, Carlsbad, CA, 37–52. https://www.usenix.org/conference/atc22/presentation/yang-zuoru

[188] Yuval Yarom and Katrina Falkner. 2014. FLUSH+RELOAD: A High Resolution, Low Noise, L3 Cache Side-Channel Attack. In *23rd USENIX Security Symposium (USENIX Security 14)*. USENIX Association, San Diego, CA, 719–732. https://www.usenix.org/conference/usenixsecurity14/technical-sessions/presentation/yarom

[189] Yuval Yarom, Qian Ge, Fangfei Liu, Ruby B. Lee, and Gernot Heiser. 2015. Mapping the Intel Last-Level Cache. *IACR Cryptology ePrint Archive* 2015 (2015), 905. https://eprint.iacr.org/2015/905

[190] Hang Yin, Shunfan Zhou, and Jun Jiang. 2019. Phala network: A confidential smart contract network based on polkadot.

[191] Peterson Yuhala, Pascal Felber, Valerio Schiavoni, and Alain Tchana. 2021. Plinius: Secure and Persistent Machine Learning Model Training. In *2021 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 52–62. https://doi.org/10.1109/DSN48987.2021.00022

[192] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of*

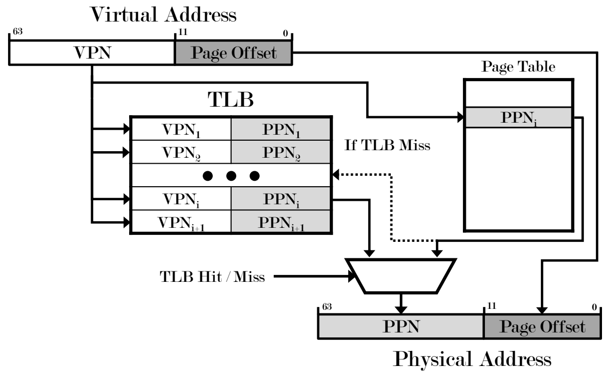


Figure 12: Virtual to physical address translation. If a TLB miss occurs, a page walk is triggered and the translation is retrieved from the page table.

the 2016 ACM SIGSAC Conference on Computer and Communications Security (Vienna, Austria) *(CCS '16)*. Association for Computing Machinery, New York, NY, USA, 270–282. https://doi.org/10.1145/2976749.2978326

[193] Wenting Zheng, Ankur Dave, Jethro G Beekman, Raluca Ada Popa, Joseph E Gonzalez, and Ion Stoica. 2017. Opaque: An Oblivious and Encrypted Distributed Analytics Platform.. In *NSDI*, Vol. 17. 283–298.

A ADDITIONAL BACKGROUND

A.1 Cache-based covert channels

Covert channels refer to communication channels that were not intended nor designed to transmit information. In particular, hardware-based covert channels constitute an effective means to establish communication between two processes that share the hardware [96]. Various micro-architectural elements can be used as a covert channel, e.g. the TLBs [176], the memory bus [184] or the DRAM [142] but, among all of them, cache memories have been the most widely studied [65, 103, 128, 134, 188]. In what follows, we briefly discuss the operation of virtual memory and caches, before providing details on cache-based covert channels as they are relevant to CloneBuster.

Virtual Memory and Cache Hierarchy. The virtual memory system is one main constituent of the memory system. In a nutshell, the virtual address space of each process is allocated in pages of typically 4KB of size, and each is stored physically in the DRAM by the OS. The OS determines the mapping between virtual and physical pages by managing a structure called "page table". Moreover, the hardware caches the mapping information of the page table in the so-called Translation Lookaside Buffer (TLB), so subsequent translations of the same virtual address are faster. Figure 12 shows how a virtual address is translated to a physical one by means of the page table. A virtual address is made of a Virtual Page Number (VPN) and a page offset (12-bits long if 4KB pages are used). The VPN is used to address the page table and obtain the corresponding Physical Page Number (PPN); the latter is concatenated with the page offset to obtain the physical address corresponding to the virtual address.

Cache memories are several orders of magnitude faster than the main memory and speed up memory access by keeping copies of recently used data. In Intel SGX, caches are shared between

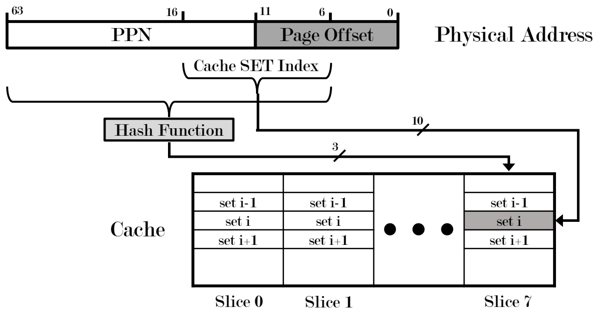


Figure 13: Representation of the cache set and slice addressing function for a cache with 8 slices and 1024 sets per slice.

enclaves and the untrusted software. Modern processors include hierarchically organized caches: L1 and L2 caches are private to each core whereas L3 cache (also known as last level cache or LLC) is shared among all the cores. Traditionally, Intel has built processors with inclusive last-level caches: data in the core-private low-level caches is also present in the shared L3 cache. The size and the latency of the caches increase as their distance to the processor grows. Indeed, by measuring the time it takes to load a piece of data, it is possible to determine whether it was located in the cache hierarchy (cache hit) or if it has been fetched from main memory (cache miss).

Intel processors include W-way set-associative caches. This type of cache is organized into multiple sets (S), each containing W lines of usually 64 bytes of data. Additionally, sets can be grouped into slices. Figure 13 shows how the hardware determines the cache location of a memory block from its physical address. The least significant bits (line offset) locate the data within the cache line, i.e., 6 bits for a 64 byte line; the following $\log_2(S)$ bits (cache set index) select the set; the slice number is determined by computing a hash function on some bits of the physical address; the remaining bits (tag) determine whether the data is cached or not. The newest generations of Intel processors include slices of 1024 sets, therefore requiring 10 bits for addressing a cache set. That is, bits 0-5 are used for the cache line, and bits 6-15 for the cache set index. If we consider 4KB pages, the application only knows 6 of the bits that determine the cache set number (bits 6-11), whereas the remaining bits (12-15) are defined by the physical address that, in turn, is chosen by the OS.

The order in which the elements of a cache set are removed from the cache and replaced with new data is determined by insertion/replacement policies [51, 62, 178]. In particular, in the case of the L3 cache, the Quad-Age Replacement Policy assigns ages ranging from 0 to 3 to the cache-lines which are updated upon accesses to the L3 cache; in the event of a cache miss the first block whose age is equal to 3 is replaced.

Cache-based Covert Channels. Cache-based covert channels exploit the noticeable timing difference between cache hits (the data is fetched from the cache) and cache misses (data has to be retrieved from main memory). In a nutshell, assuming a sender and a receiver, the receiver first sets the cache into a known state (e.g., by forcing some data into the cache), then the sender would either remove that data (e.g., by either flushing the cache or forcing some new data into the cache so to ensure that the original data from the receiver is removed from the cache) to send a "1", or do nothing to send a "0". Finally, the receiver checks (e.g., by probing the cache) whether the cache has changed from its original state.

Using the L3 cache as a covert channel requires building eviction sets, which are groups of memory addresses mapping to a particular set and slice [179]. These are the main building blocks for Prime+Probe attacks [128, 137, 160] or for achieving fast evictions required to carry out rowhammer attacks [92]. For regular applications, large memory pages of 2MBs ease the creation of eviction sets. Unfortunately, large pages are not available for Intel SGX enclaves. However, Islam et al. [105] have shown how to leverage the processor load and store operations to tell whether two addresses share up to 20 bits; this technique can be used to build eviction sets even in case large pages are not available.

A.2 DRAM

We now provide additional background information on the DRAM and how it can be used to infer information on the physical memory allocated to an enclave [160, 161]. This information will be particularly useful for Appendix B.

The DRAM, sometimes referred to as main memory, is organized hierarchically: the memory arrays arranged in rows and columns form banks which, in turn, form bank groups in the case of DDR4. A set of banks composes a rank (typically 8 banks on DDR3 and 16 banks on DDR4). Each memory cell is uniquely addressed with the information referring to its channel, DIMM, rank, bank, row and column. The CPU maps physical memory addresses to cells by means of an (officially) undocumented function. Previous work has reverse-engineered the function [142] and showed that two alternating accesses to the DRAM mapping to the same bank (i.e., they have the same value of BA0, BA1, BG0 and BG1, rank, and channel) but that have different row indexes (row conflict) would take considerably longer than if they had the same row index (row hit). By timing memory accesses, a process can infer relations among some bits of its physical address space.

We reverse-engineered the mapping function for our Xeon E-2176G (2x32GB DDR4 RAM) as shown in [142]. The function is depicted in Figure 14. In a nutshell, given a physical address as $a_{n-1}, a_{n-2}, ..., a_0$, our memory controller computes its channel as $a_{18} \oplus a_{15} \oplus a_{13} \oplus a_{12} \oplus a_9 \oplus a_8$; the bank values BG0, BG1, BA0, and BA1 as $a_{19} \oplus a_{15}$, $a_{20} \oplus a_{16}$, $a_{21} \oplus a_{17}$, and $a_{22} \oplus a_{18}$, respectively; the rank as $a_{22} \oplus a_{18}$. Finally, the row index uses bits 18 and above from the physical address (assuming a typical row size of 8KB).

Note that Figure 14 does not show bits 0 to 5 since they are used neither to address the DRAM nor the cache sets. We use different background colors to distinguish between regions: the region between bit 6 and bit 11 represents the bits of the cache set index that are controlled by the enclave; the region between bit 12 and bit 15 (blue) corresponds to the remaining bits of the cache set index under control of the OS. Additionally, the region between bits 16 and 19 (gray) shows the bits used by Spoiler [105] that are not part of the cache set index; finally, the region that goes from bit 20 to bit 22 are the bits exclusively used by the DRAM mapping function.

	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6
Channel					X			X		X	X			X	X		
BG0									X							X	
BG1				X				X									
BA0			X				X										
BA1		X				X											
Rank	X				X												

Figure 14: DRAM mapping function of our test machine (Xeon E-2176G). Each output bit value (row) is computed XOR-ing the bits of the physical address (columns) marked with an *X*.

Algorithm 2 Encoding of the conditions for the SAT Solver

```
Require: Set of conditions on arr[N][B]
Ensure: Mapping that fulfills the conditions
// Setup phase (assign known values)
c1=(), c2=(),c3=(), c4=(), c5=()                                  ▷ Initialize conditions
for $i$ = 0 to N do
    arr[i][0:11] = i mod $2^{12}$ (4096)
// Condition 1
for $i$ = 0 to N do
    for $j$ = $i$ to N do
        c1 &= (at least one bit of arr[i][:] different from arr[j][:])
// Condition 2
for $i$ = 0 to $2^{20}$ do
    for $j = i + 256$ to N; $j += 2^{20}$; do
        c3 &= (arr[i][0:19] equal to arr[j][0:19])
// Condition 3
for $i$ = 0 to $2^{16}$ do
    for $j = i + 16$ to N; $j += 2^{16}$; do
        c3 &= (arr[i][0:15] equal to arr[j][0:15])
// Condition 4
for $i$ = 0 to $2^6$ do                   ▷ i ≡ {channel,BG0,BG1,BA0,BA1,Rank}
    $list \leftarrow ()$
    for $j$ = 0 to N do
        $channel_j = j_{18} \oplus j_{15} \oplus j_{13} \oplus j_{12} \oplus j_9 \oplus a_8$;
        $BG0_j = j_{19} \oplus j_{15}$
        $BG1_j = j_{20} \oplus j_{16}$
        $BA0_j = j_{21} \oplus j_{17}$
        $BA1_j = j_{22} \oplus j_{18}$
        $Rank_j = j_{22} \oplus j_{18}$;
        if i == {$channel_j, BG0_j, BG1_j, BA0_j, BA1_j,$
$Rank$} then
            Add j to $list$
    for all k in list do
        for all l in list such l!= k do
            if $k_{18:B}! = l_{18:B}$ then                              ▷ Row bits
                c4 &= arr[k][:] conflicts with arr[l][:]
            else
                c4 &= arr[k][:]                                      ▷ No conflict
// Condition 5
c5 &= arr[0][:] has row conflict with arr[N-1][:])
for $i$ = 0 to N-1 do
    if i then mod $2^{22}$ != (0x7FFFFF)
        c5 &= arr[i][:] has no conflict with arr[i+1][:])
    else
        c5 &= arr[i][:] conflicts with arr[i+1][:])
Expression to satisfy = c1 & c2 & c3 & c4 & c5
```

B LINEAR MEMORY MAPPING AND PHYSICAL ADDRESSES

To build eviction sets for CloneBuster, we opted not to rely on the technique proposed in [160, 161] due to its reliance on an OS that assigns linear memory to enclaves. We now provide more details on this choice. In particular, we show that a malicious OS can assign to an enclave a non-linear memory region that "looks" linear from the enclave perspective. As a consequence, if eviction sets are built by assuming linear memory, a malicious OS can deceive two enclave clones to monitor different cache sets—so that they would not detect each other.

Recall that [160] builds evictions sets by relying on the assumption that the physical memory assigned to an enclave is linear. In particular, if the memory is linear, given a virtual address VA_i and its corresponding physical address PA_i, the physical address of another virtual address VA_j is computed as $PA_j = PA_i + (VA_j - VA_i)$. As a consequence, given any two addresses VA_i, VA_j such that $VA_i[15-0] = VA_j[15-0]$, then $PA_i[15-0] = PA_j[15-0]$ and the two addresses share the same cache set. In this setting, it is straightforward to find an eviction set for a specific cache set, if the value of $PA_i[15-0]$ is known. In case the assigned memory is linear, the enclave can find out bits 15-0 of a physical address PA_i corresponding to a virtual address VA_i by exploiting the DRAM mapping function. In particular, a process witnesses a DRAM row conflict between two consecutive virtual addresses VA_i and $VA_i + 1$ only if their physical addresses have their last 22 bits equal to all ones and all zeros, respectively [160, 161]. Thus, a process can cycle through its virtual memory to find two virtual addresses VA_i and $VA_i + 1$ such that their corresponding physical addresses have $PA_i[15-0] = \texttt{0xFFFF}$ and $PA_{i+1}[15-0] = \texttt{0x0000}$, respectively. Once those physical addresses have been found, either can be used as a starting address to build an eviction set.

In our scenario, the OS is however considered malicious and has an obvious advantage in assigning non-linear memory to the victim enclave. In particular, assigning non-linear memory to two enclave clones may deceive them to monitor different cache sets—and therefore not detect each other—as follows. Given two enclave instances $E1$ and $E2$ the OS assigns the physical memory of $E1$ such that, when $E1$ performs the above DRAM test, it finds two consecutive physical addresses such that $PA_i[15-0] = \texttt{0xFFFF}$ and $PA_{i+1}[15-0] = \texttt{0x0000}$. Assume $E1$ uses PA_{i+1} as the starting point to build an eviction set so that the monitored cache set has cache set number 0 (because $PA_i[15-6]$ is all zeros). Now the OS "swaps" the two addresses for $E2$ so that, when running the above test, $E2$ finds $PA_i[15-0] = \texttt{0x0000}$ and $PA_{i+1}[15-0] = \texttt{0xFFFF}$. Thus, if $E2$ uses PA_{i+1} as the starting point to build an eviction set, it ends up monitoring cache set number 1023 (because $PA_i[15-6]$ is all ones) and the two enclaves will not detect each other.

Since we cannot trust the OS to assign contiguous linear memory to enclaves, building eviction sets using [160] requires checking that the memory assigned to the enclave is indeed linear. To do so the enclave could leverage known techniques to infer information of the physical memory corresponding to its virtual memory region. In particular, (i) speculative loads [105] allow the enclave to check whether two addresses share the least significant 20 bits, (ii) cache memory allows the enclave to verify if groups of addresses share the least significant 16 bits, and (iii) the DRAM provides information about the bits of the physical addresses by measuring row hits and conflicts—as explained earlier.

Given the above techniques, we define a set of conditions that a linear memory arrangement must meet:

Condition 1 All physical address, to which virtual addresses are mapped, must be different.

Condition 2 Given two virtual addresses sharing the last 20 bits, we must observe speculative load hazards [105].

Condition 3 Given two virtual addresses sharing bits 6-15, they must be assigned to the same cache set.

Condition 4 Given any virtual address VA_i for which we assume its physical address is known, if we calculate its corresponding channel, BG0, BG1, BA0, BA1, Rank and row value we must observe row conflicts with any other VA_j whose physical address (obtained assuming there is linear memory) that maps to the same channel, BG0, BG1, BA0, BA1 and Rank while having a different row value. Analogously, we must observe row hits if the row value is the same.

Condition 5 For any VA_i and VA_j such that $VA_j = VA_i + 1$, we should observe a row conflict if and only if PA_i has its 22 least significant bits equal to 1 and consequently PA_j has its 22 least significant bits equal to 0 (e.g., $VA_i = \texttt{0x3FFFFF}$ and $VA_j = \texttt{0x400000}$)

Ideally, if an enclave checked the conditions above and if a linear memory arrangement were the only arrangement to fulfill such conditions, we could build eviction sets as in [160], while being absolutely sure about the value of the bits 12-15.

To verify the existence of alternative (i.e., non-linear) memory assignments, we formulate this problem as a Boolean satisfiability problem (SAT). We encode each bit of the physical addresses of a memory range as a condition and then use a SAT solver to figure out the possible values of such physical addresses that satisfy these conditions. Concretely, we used SATisPy [89], which is a wrapper of the popular MiniSAT [82]. For reproducibility purposes, the encoding of the problem is given in Algorithm 2. SATisPy provided multiple non-linear memory arrangements that satisfied all of the conditions. We conclude that building eviction sets by using the technique of [160] may not be suitable for CloneBuster, as the OS may provide a page alignment that would help it to place two clone enclaves on two different channels and evade detection. For this reason, CloneBuster builds eviction sets by using a technique derived from [105]. Also, because of a malicious OS, CloneBuster may not be able to figure out bits 12-15 of one of its physical addresses and, therefore, cannot be sure of its cache set index. For this reason, CloneBuster uses the 16 possible cache sets as a covert channel—fixing bits 6-11 of the addresses of the eviction sets—and monitors all of them to make sure that the OS cannot evade detection.

The code in Algorithm 2 uses an array $arr[N][B]$ that mimics the region of the virtual memory we want to study in the sense that it encodes N different addresses of B bits. The actual content of each location of the array represents the physical address, whereas the location itself represents the virtual address. In particular, we used a memory region that comprises 8MB because the DRAM function uses up to the 23rd bit. The conditions are coded as if $arr[0][:]$ was going to have its content equal to 0, i.e., they assume that the physical address of $arr[0][:]$ is $\texttt{0x000000}$.

C ADDITIONAL EXPERIMENTS

In Table 4, we report for completeness, the evaluation results when CloneBuster is equipped with other detection algorithms; Table 4 also shows the performance of CloneBuster when different applications of the Phoronix benchmark suite [120] are running in the

		Logistic regression						Linear Discriminant Analysis						KNN					
		w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024
Baseline	m=9	0.884	0.936	0.973	0.992	0.991	0.986	0.884	0.936	0.957	0.979	0.983	0.993	0.885	0.925	0.961	0.954	0.973	0.989
	m=12	0.906	0.965	0.996	0.977	0.983	0.990	0.906	0.966	0.985	0.986	0.991	0.993	0.907	0.968	0.992	0.993	0.992	0.993
	m=16	0.829	0.894	0.967	0.977	0.984	0.985	0.829	0.894	0.957	0.977	0.977	0.978	0.746	0.885	0.965	0.977	0.980	0.986
x265	m=9	0.801	0.870	0.912	0.922	0.951	0.961	0.801	0.870	0.888	0.900	0.952	0.957	0.719	0.878	0.925	0.931	0.954	0.959
	m=12	0.907	0.959	0.972	0.973	0.974	0.983	0.907	0.953	0.971	0.971	0.971	0.972	0.363	0.957	0.976	0.978	0.979	0.984
	m=16	0.757	0.801	0.829	0.901	0.952	0.959	0.757	0.801	0.827	0.897	0.912	0.937	0.227	0.780	0.849	0.892	0.914	0.939
sql	m=9	0.894	0.919	0.938	0.944	0.973	0.989	0.894	0.908	0.914	0.936	0.959	0.971	0.893	0.923	0.940	0.941	0.944	0.939
	m=12	0.945	0.971	0.978	0.984	0.990	0.994	0.945	0.969	0.978	0.980	0.975	0.987	0.945	0.973	0.978	0.983	0.987	0.987
	m=16	0.812	0.841	0.849	0.875	0.932	0.980	0.812	0.839	0.848	0.870	0.924	0.978	0.321	0.849	0.853	0.886	0.905	0.934
opencv	m=9	0.856	0.888	0.925	0.935	0.947	0.955	0.855	0.889	0.924	0.945	0.956	0.965	0.855	0.888	0.934	0.946	0.964	0.967
	m=12	0.903	0.930	0.937	0.956	0.963	0.974	0.903	0.930	0.936	0.945	0.955	0.963	0.903	0.930	0.938	0.949	0.952	0.961
	m=16	0.670	0.775	0.876	0.900	0.911	0.917	0.670	0.775	0.868	0.905	0.910	0.915	0.670	0.775	0.921	0.925	0.923	0.911
gcc	m=9	0.850	0.909	0.933	0.938	0.960	0.979	0.850	0.909	0.917	0.937	0.939	0.946	0.848	0.907	0.938	0.954	0.961	0.962
	m=12	0.931	0.973	0.980	0.982	0.985	0.987	0.931	0.967	0.980	0.983	0.987	0.989	0.931	0.970	0.979	0.979	0.982	0.978
	m=16	0.809	0.838	0.873	0.887	0.923	0.933	0.809	0.839	0.850	0.864	0.906	0.923	0.559	0.835	0.870	0.900	0.933	0.943
cloud	m=9	0.933	0.967	0.979	0.987	0.990	0.990	0.933	0.955	0.955	0.959	0.963	0.971	0.932	0.970	0.974	0.985	0.986	0.988
	m=12	0.906	0.941	0.953	0.964	0.987	0.990	0.906	0.929	0.929	0.9423	0.954	0.973	0.901	0.942	0.944	0.959	0.983	0.989
	m=16	0.856	0.916	0.932	0.939	0.941	0.949	0.856	0.908	0.932	0.946	0.957	0.972	0.256	0.918	0.934	0.943	0.951	0.966

		Decision Tree						Random Forest						SVM					
		w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024	w=1	w=4	w=16	w=64	w=256	w=1024
Baseline	m=9	0.884	0.936	0.982	0.989	0.988	0.990	0.884	0.936	0.963	0.973	0.979	0.982	0.886	0.931	0.977	979	0.980	0.981
	m=12	0.906	0.970	0.998	0.992	0.993	0.991	0.906	0.970	0.995	0.996	0.994	0.993	0.907	0.969	0.993	0.993	0.993	0.993
	m=16	0.829	0.895	0.980	0.981	0.989	0.992	0.829	0.895	0.956	0.978	0.985	0.991	0.830	0.901	0.963	0.979	0.978	0.986
x265	m=9	0.801	0.895	0.948	0.966	0.970	0.973	0.801	0.895	0.946	0.957	0.963	0.973	0.798	0.894	0.952	0.966	0.974	0.980
	m=12	0.907	0.960	0.971	0.971	0.954	0.982	0.907	0.960	0.972	0.978	0.983	0.992	0.906	0.959	0.972	0.970	0.973	0.980
	m=16	0.757	0.792	0.848	0.912	0.944	0.951	0.757	0.792	0.841	0.916	0.952	0.969	0.755	0.784	0.845	0.920	0.931	0.920
sql	m=9	0.894	0.923	0.964	0.969	0.972	0.972	0.894	0.923	0.951	0.968	0.954	0.962	0.893	0.924	0.961	0.983	0.990	0.992
	m=12	0.945	0.973	0.977	0.978	0.980	0.981	0.945	0.973	0.978	0.980	0.981	0.983	0.945	0.971	0.974	0.968	0.978	0.979
	m=16	0.812	0.846	0.859	0.899	0.927	0.952	0.812	0.846	0.853	0.938	0.941	0.944	0.811	0.846	0.854	0.875	0.955	0.978
opencv	m=9	0.855	0.888	0.933	0.959	0.960	0.963	0.855	0.888	0.918	0.931	0.960	0.967	0.855	0.889	0.931	0.943	0.935	0.939
	m=12	0.903	0.930	0.934	0.943	0.955	0.969	0.903	0.930	0.937	0.954	0.959	0.973	0.903	0.928	0.939	0.960	0.967	0.969
	m=16	0.670	0.775	0.957	0.947	0.971	0.970	0.670	0.775	0.868	0.911	0.927	0.963	0.667	0.774	0.890	0.917	0.963	0.971
gcc	m=9	0.850	0.914	0.951	0.958	0.965	0.973	0.850	0.914	0.952	0.962	0.965	0.969	0.849	0.915	0.957	0.962	0.962	0.973
	m=12	0.931	0.973	0.975	0.981	0.982	0.983	0.931	0.973	0.979	0.980	0.982	0.982	0.931	0.971	0.979	0.981	0.986	0.986
	m=16	0.809	0.839	0.896	0.917	0.926	0.937	0.809	0.839	0.869	0.920	0.946	0.972	0.806	0.841	0.890	0.915	0.915	0.920
cloud	m=9	0.933	0.967	0.984	0.986	0.989	0.990	0.933	0.967	0.977	0.986	0.987	0.987	0.933	0.966	0.984	0.986	0.989	0.992
	m=12	0.906	0.941	0.960	0.982	0.983	0.983	0.906	0.941	0.952	0.972	0.980	0.9832	0.907	0.943	0.957	0.977	0.993	0.996
	m=16	0.856	0.921	0.944	0.951	0.966	0.972	0.856	0.921	0.934	0.941	0.948	0.967	0.857	0.920	0.935	0.966	0.987	0.989

Table 4: F1 score of various detection algorithms for different values of w and m. Baseline refers to the scenario where no background applications are running, whereas the others refer to different applications running in the background simultaneously with the clone detector.

background. Our results reveal a performance degradation when $m = 16$—that is, when monitoring all of the lines in the 16 cache sets chosen as the covert channel—and an application is running in background. Finally, we note that there is no clear choice between $m = 9$ and $m = 12$.

D ALTERNATIVE ARCHITECTURES

Other TEEs. CloneBuster has been mostly designed for Intel SGX. Nevertheless, it also finds direct applicability in a number of other TEE instantiations. In what follows, we discuss how CloneBuster can be adapted to other TEE architectures.

Recall that the main requirement for CloneBuster to work is the existence of a resource shared among clones that can be used as a covert channel. We argue that if a platform is vulnerable to covert-channels, this vulnerability could be turned into a feature to detect clones. Previous work has shown a number of micro-architectural features on different platforms that could be used to setup side- or covert-channels (see [88] for a survey). In the following, we provide more details on ARM TrustZone and AMD SEV—arguably the most popular TEEs along with Intel SGX.

ARM TrustZone [143] is the TEE instantiation proposed by ARM and distinguishes between two protection domains dubbed secure world (SW) and normal world (NW). Previous work has shown how normal world software can instruct the OS in the trusted world to launch multiple instances of the same process [53, 117]. Hence, applications running in the secure world, could leverage a clone-detection mechanism like CloneBuster, by exploiting one of the available covert-channel [63, 72, 126]. Notice, however, that details of ARM TrustZone vary across implementations and the design of a particular anti-cloning mechanism for ARM TrustZone should take into account the specific feature of the underlying ARM platform.

Similarly, AMD Secure Encrypted Virtualization (SEV) [78] aims to protect virtual machines from untrusted cloud providers or hypervisors. In a nutshell, SEV includes an encryption engine that automatically encrypts and decrypts data in main memory to ensure that it is not readable by other software on the same platform. According to its documentation, SEV does not offer any mechanism to prevent a malicious hypervisor from launching several clones of a victim VM. Further, unless the VM owner sets its VM as non-migratable (by setting the NOSEND bit in guest policy), the hypervisor can clone enclaves across platforms, thereby making the problem of detecting clones even tougher to solve. Assuming that migration is disabled— thus the adversary can only clone the victim VM on the same platform— a VM could detect its clones on the same machine by using a covert channel as shown by CloneBuster. We note AMD processors are vulnerable to side- and covert-channels [127], and also the official AMD documentation acknowledges that SEV VMs are vulnerable to Prime+Probe attacks [77]. We also note that the way AMD SEV partition caches among guests—guests sharing the same Address Space Identifier (ASID) are assigned to the same cache sets—may facilitate the creation and improve the throughput of the covert channel between clones; this is because two different VMs (with different ASIDs) will not use the same cache sets, hence they will not pollute each-other's cache.

Non-inclusive caches. CloneBuster requires the enclave to build eviction sets on the host, despite a potentially malicious OS. We have shown how to do so on a CPU architecture with inclusive caches, since this is the cache architecture available on all SGX-enabled CPUs to date. Nevertheless, Intel has recently announced that SGX will also be available on next-generation Xeon processors [102], which most likely will feature non-inclusive caches and increase the size of the EPC. In order to use CloneBuster on such processors, we could leverage techniques to build eviction sets on architectures with non-inclusive caches as proposed by Yan et al. [185]. In particular, one could build eviction sets by targeting cache directories rather than the cache itself.

Cache Allocation Technology. Cache Allocation Technology (CAT) [98] allows the OS or any privileged software to define multiple partitions in the LLC. In a nutshell, CAT introduces an intermediate abstraction, so-called Class of Service (CLOS), into which applications can be grouped. The CLOS is in turn associated with a capacity bitmask (CBMs), which indicates the fraction of the LLC that can actually be used by the given CLOS [94].

If future SGX-enabled processors were to support CAT[4], CloneBuster could easily detect if it is only assigned to one partition of the cache since, in this case, it will fail to build eviction sets. In this case, the application may take appropriate countermeasures, e.g., refuse to execute. In principle, a malicious OS could try to use CAT dynamically: it could let CloneBuster build eviction sets while accessing the whole cache, and later on restrict each clone to a different cache partition. We speculate that such a strategy would cause false positives (i.e., each enclave would signal the presence of clones) as the enclave will evict its own data when trying to fill the monitored sets. This is due to the fact that some of the data will not fit in the newly assigned sets or slices (physical addresses do not change and we monitor all the slices and expand through multiple sets).

[4]Notice that CAT is not available on SGX-enabled processors [100].

E CLONING ATTACKS

We have examined 72 application based on Intel SGX with respect to forking attacks (see Section 3). Among the examined applications, 14 were susceptible to cloning-based forking attacks. In what follows, we describe in details how to mount cloning attacks on these applications. We observe that cloning-based attacks can be grouped into three broad categories. To ease the presentation, we describe each of the attack categories and, for each category, we show how the attack can mounted against an exemplary applications (chosen from the 14 vulnerable applications).

E.1 FIm – Forking In-memory Key-value Stores

We start by describing cloning attacks on in-memory key-value stores (KVS), dubbed FIm. A KVS exposes PUT and GET interfaces to clients. At any time, there is at most one value per key; further, when a GET request is issued for a given key k, the latest value that was written to the database for that key is returned. A forking attack against a KVS may break these security guarantees.

In-memory KVSs that use Intel SGX, e.g., Aria [186], Enclage [169], STANLite [154], ObliDB [83], and Avocado [55], are designed to store data larger than EPC memory (limited to 128 MB). Hence, they seal data to persistent memory but keep meta-data in runtime memory to ensure integrity and rollback protection. Since meta-data is not sealed to persistent storage, it is lost if the enclave terminates.

E.1.1 Attack overview. In benign settings, assume a server running an enclave-backed KVS that two clients, A and B, can access as depicted in Figure 15. First, both clients attest the enclave and establish a session key to encrypt their messages. All subsequent messages are encrypted using the session key. In a benign setting, A sends a PUT request to post the key-value (KV) pair (k, v_A) to the storage. The enclave recognizes that the key k does not exist, creates a new entry with the pair (k, v_A), and returns an *ACK* message. Afterward, B sends a PUT request to post the KV pair (k, v_B). At this time, the enclave recognizes that the key k exists and updates the value to v_B before returning an *ACK* message. If A later requests the value associated with k from the KVS, it receives the value v_B.

In an adversarial setting (cf. Figure 16), the adversary can provide two different KVS instances to A and B. Even if clients attest the enclave where the KV instance is running, they cannot tell whether they are communicating with the same instance or not.

The adversary launches two enclave instances, E_A and E_B, and connects each client to one instance. Each client attests the connected enclave. Assume the same sequence of requests as in the previous setting. First, A sends a PUT request to post the KV pair

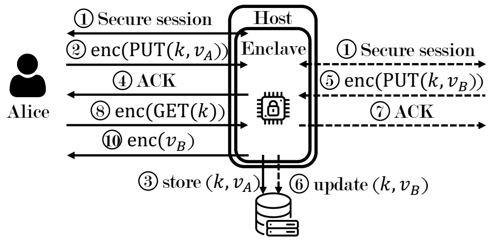


Figure 15: Overview of an SGX-backed in-memory key-value store if it operates in a benign setting.

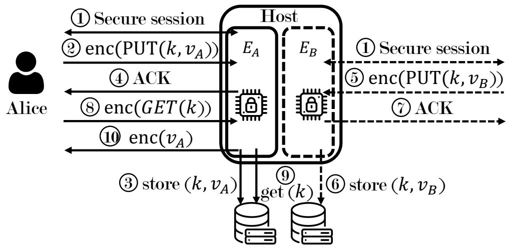


Figure 16: Overview of a generic FIm attack on an SGX-backed in-memory key-value store.

(k, v_A) to the storage. E_A recognizes that k does not exist in the associated storage, creates a new entry with the pair (k, v_A), and returns an *ACK* message. Afterward, client B sends a PUT request to post the KV pair (k, v_B). E_B does not find an entry for k in its copy of the storage and creates a new entry for the pair (k, v_B). Both clients receive an *ACK* reporting the correct execution of their request. However, if client A later requests the value associated with k, E_A returns v_A which is the latest value it has seen—this is however different from the newest value in the system.

We note that the above issue can be fixed if clients are mutually trusted and each client shares its view of the KVS with the others [61]. Essentially, the set of client acts as a distributed TTP. Unfortunately, assuming a mutually trusted set of clients limits the scenarios where the KVS can be used.

E.1.2 Concrete example: FIm Attack against Aria. As an example, we now show how to mount a FIm attack against **Aria** [186]. Aria provides an in-memory KVS in the cloud. Each entry is protected against rollback attacks by means of MC. For confidentiality, the entries are encrypted with AES (in CTR mode), where the counter value is set to be the current MC value of the entry. The enclave generates a pseudo-random key at initialization and uses the same key for encrypting all data. Additionally, each entry contains a MAC over the encrypted data for integrity protection. The integrity of the MCs is guaranteed by a Merkle tree structure over all MCs. The enclave exclusively stores the Merkle root in its runtime memory. Additionally, it stores all recently used MCs in its local cache. The cached counters can be used to decrypt entries directly without verifying the Merkle root, thus reducing latency.

As depicted in Figure 17, the client first attests the enclave and establishes a secure session key. The client sends a PUT request for (k, v), encrypted with the session key. The enclave decrypts

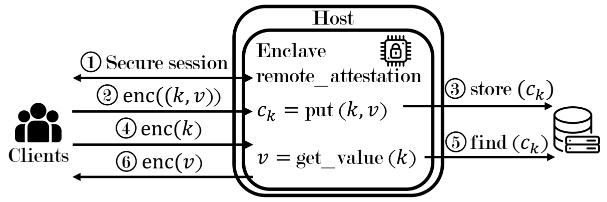


Figure 17: Overview of the main functions exposed by the Aria enclave and its interactions with clients.

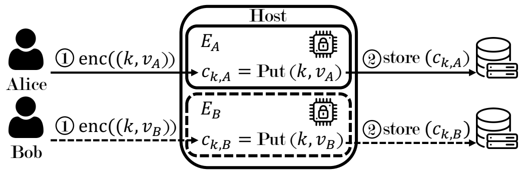


Figure 18: Overview of a cloning attack against Aria enclaves.

the message and checks if k exists in storage. If so, it updates the corresponding counter and encrypted KV pair. Otherwise, it assigns the key a free counter and stores it in the database. Later, the client can access v by sending an encrypted GET request for k. The enclave verifies the counter integrity and decrypts entries until it finds the requested KV pair. Finally, it returns v through the secure channel.

Assume now a malicious host and two clients, A and B who share access to the same KVS, e.g. for customer records.

As shown in Figure 18, one can mount FIm attacks on Aria as follows:

- The adversary starts two Aria enclave instances, E_A and E_B.
- The adversary connects A to E_A, and B to E_B.
- The clients attest the enclaves and establish secure communication sessions.
- The clients send encrypted PUT requests for (k, v_A) and (k, v_B) to E_A and E_B, respectively.
- E_A and E_B decrypt the requests and create/update the corresponding encrypted entries in the their storage instances. A and B cannot determine if they are communicating with the same instance.

Hence, FIm violates the consistency of Aria by cloning the enclave. The adversary is not limited by the number of enclaves and can run arbitrarily many instances.

E.2 ForKVS – Forking persistent Key-value Stores

We now describe cloning attacks on persistent KVSs, dubbed ForKVS. Persistent KVSs that are susceptible to cloning attacks are EnclaveCache [67], NeXUS [76], StealthDB [180], ShieldStore [111], SGX-KMS [64], and CACIC [170]. In contrast to in-memory KVSs, persistent KVS use sealing to make meta-data available across reboots.

E.2.1 Attack overview. Assume a server running an enclave-backed KVS that stores a KV pair (k, v_0) when a client, C, connects to the system. First, C attests the enclave E_C and establishes a session key.

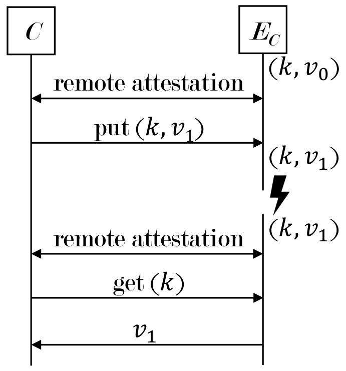


Figure 19: Overview of the interaction of a persistent key-value store with a client in a benign setting.

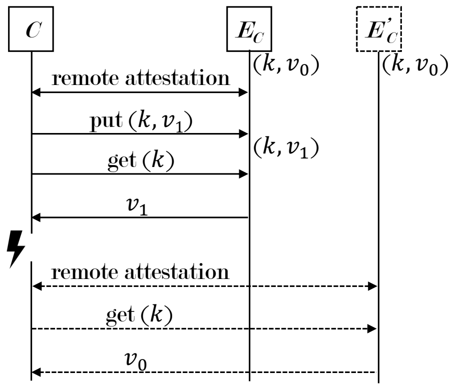


Figure 20: Overview of a generic ForKVS attack against persistent key-value stores.

All subsequent messages are encrypted using the session key. In a benign setting (cf. Figure 19), C sends a PUT request to update the value associated with k to v_1. E_C updates the KV pair in its storage. Afterward, it creates a snapshot, sealing the meta-data with the incremented MC value $ctr_i + 1$ and incrementing the MC value $ctr_{i+1} \leftarrow ctr_i + 1$. Later, E crashes and needs to restart. It successfully verifies the MC value in the sealed data and restores the KVS. C must attest the restarted enclave instance and establish new session keys. When C requests the value associated with k, the KVS correctly returns the latest value, v_1.

In an adversarial setting (cf. Figure 20), the adversary can provide two different views of the same KVS instance to C. The adversary launches two enclave instances, E_C and E'_C. Both instances have the same initial state storing the KV pair (k, v_0). First, the adversary connects C to enclave E_C, and the value is updated to v_1. If the client requests the value associated to k, the enclave correctly provides with v_1. Afterward, the adversary connects C to the second instance, E'_C. C assumes the enclave has crashed and successfully attests E'_C. However, when requesting the value associated with k, E'_C returns v_0, the latest state it stores. Consequently, the same key is associated with different values in different enclave instances. The same attack holds if multiple clients use the KVS instead of C connecting to the KVS in different sessions.

Cloning attacks on in-memory KVSs are limited to providing two instances of a KVS. They do not share entries unless the same data is provided to both instances in different sessions. ForKVS is more powerful: two instances of the KVS share common data that has been sealed by the first instance before the second instance starts. Therefore, ForKVS can have the same effect as rollback attacks, even if classical rollback attacks are not possible.

Notice that BI-SGX can be seen as an instantiation of a persistent KVS. We therefore refer the reader to Section 3.3 as a concrete example of ForKVS.

E.3 BUG – Breaking Unlinkability Guarantees

We now describe cloning attacks on SGX proxies, dubbed BUG. Applications affected by BUG, i.e., X-Search [138] and PrivaTube [73], provide unlinkability by leveraging an SGX-backed proxy. The proxy receives encrypted requests and obfuscates them, e.g., by adding fake requests, to ensure that an adversary accessing the service cannot link the plaintext requests to individual clients.

E.3.1 Attack overview. By cloning the enclave, an adversary can break the unlinkability and link a request to a specific user or at least reduce the anonymity set. We now describe the generic cloning attack for breaking unlinkability guarantees of SGX-backed proxies, considering an honest setting first.

Assume a server running an enclave-backed proxy that receives requests from two clients, A and B, as depicted in Figure 21. First, both clients attest the enclave and establish session keys. In a benign setting, clients send requests req_A and req_B, encrypted with the session key. The enclave decrypts the requests and forwards two (decrypted) requests, req_1 and req_2, without knowing the identity of the issuer. Afterward, the proxy maps the responses to the client requests, encrypts and forwards them to the corresponding client. The server cannot distinguish if A send req_1 or req_2. The anonymity set increases with the number of clients simultaneously connected to the enclave.

In an adversarial setting (cf. Figure 22), the adversary can recover the assignment and break the unlinkability guarantee. The adversary starts two proxy enclaves, E_A and E_B, and connects clients A and B to one instance each. The clients attest the connected enclave and send the encrypted requests. The adversary observes which enclave forwards the request to the server, e.g., E_A sends the request req_1. The adversary connected A to E_A, thus can infer that A sent req_1. Linking the decrypted requests to the clients, the BUG attack breaks the unlinkability guarantee.

E.3.2 Concrete example: BUG Attack against PrivaTube proxies. As an example, we now show how to mount a BUG attack against **PrivaTube** [73]. PrivaTube is a distributed Video on Demand system leveraging fake requests and SGX enclaves to ensure the unlinkability of requests to individual users. Requests for video segments can be served by video servers and assisting platforms. Assisting platforms are other users that requested a specific video segment in the past and can provide other users with this segment. Each peer in the system hosts an enclave, an HTTP proxy, to break the link between clients and requests.

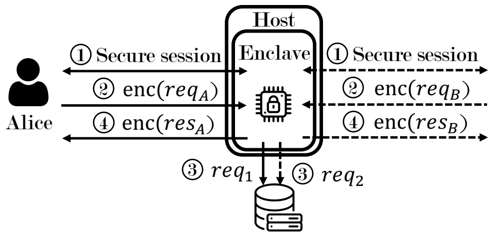


Figure 21: Overview of an SGX-backed proxy in a benign setting.

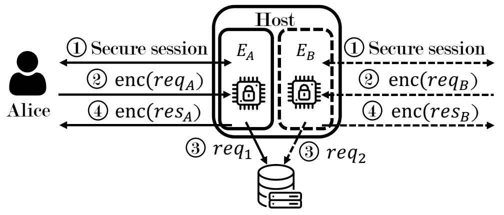


Figure 22: Overview of a generic BUG attack against an SGX-backed proxy.

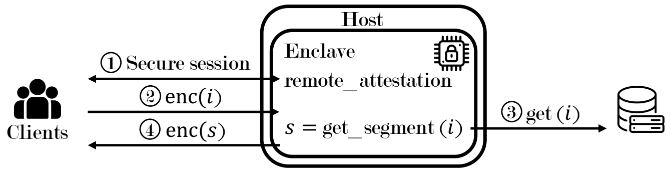


Figure 23: Overview of the functions exposed by a PrivaTube proxy and its interaction with clients.

As shown in Figure 23, a client attests the proxy enclave and sends an encrypted request for a video segment with the ID i. The enclave decrypts the segment ID and requests the video segment from the peer's video database. It encrypts the received segment s and sends it to the client. PrivaTube assumes that each video server serves multiple requests simultaneously, thus preventing the precise assignment of users to requested video segments.

Assume now a malicious video server and two users, A and B. As shown in Figure 24, one can mount a BUG attack on PrivaTube proxies as follows:

- The adversary starts two proxy enclave instances, E_A and E_B.
- The adversary connects A to E_A, and B to E_B.
- The clients attest the enclaves and establish secure communication sessions.
- The clients send encrypted requests for i_A and i_B to E_A and E_B, respectively.
- The adversary observes the decrypted requests for i_A and i_B to the database, issued by E_A and E_B, respectively.
- Knowing A is connected to E_A and B is connected to E_B, the adversary can recover that A requested the video segment i_A and B requested i_B. Both requests are served correctly. Further, A and B cannot determine that they are connected to different proxies.

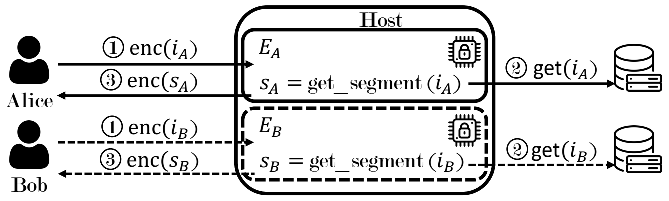


Figure 24: Overview of a cloning attack against a PrivaTube proxy.

The adversary is not limited by the amount of enclaves it can execute at the same time. For every client requesting the video server, the adversary can start a new enclave, precisely recovering the assignment of requested video segments to clients. Here, the unlinkability guarantee is broken.

Notice that BUG attacks can be mounted against other proposals (not present in the lists we have analyzed) that use Intel SGX to cloak client requests. For example, Prochlo [58] and subsequent work (e.g., [156]) provide real-world privacy-preserving analytic frameworks to collect anonymous statistics. They use Intel SGX to shuffle client inputs so to break the link between a data item and the identity of the client where data originates. In a nutshell, an external observer cannot tell the client that sent a given input, among a set of k clients. We note that, by cloning the shuffler enclave, an adversary can split the set of clients in disjoint subsets, so that two clients of two different subsets send their inputs to two different instances of the shuffler enclave. As a result, each client input is shuffled with less than k data items and matching an input to its client becomes an easier task.