# No Forking Way: Detecting Cloning Attacks on Intel SGX Applications

Samira Briongos
samira.briongos@neclab.eu
NEC Laboratories Europe
Germany

Ghassan Karame
ghassan@karame.org
Ruhr-Universität Bochum
Germany

Claudio Soriente
claudio.soriente@neclab.eu
NEC Laboratories Europe
Spain

Annika Wilde
annika.wilde@rub.de
Ruhr-Universität Bochum
Germany

## 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.

## CCS CONCEPTS

• **Security and privacy** → *Software security engineering*; *Side-channel analysis and countermeasures.*

## KEYWORDS

Trusted Execution Environments, Intel SGX, Cloning Attacks


*ACSAC '23, December 4–8, 2023, Austin, TX, USA*

ACM ISBN 979-8-4007-0886-2/23/12.
https://doi.org/10.1145/3627106.3627187

**ACM Reference Format:**
Samira Briongos, Ghassan Karame, Claudio Soriente, and Annika Wilde. 2023. No Forking Way: Detecting Cloning Attacks on Intel SGX Applications. In *Annual Computer Security Applications Conference (ACSAC '23), December 4–8, 2023, Austin, TX, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3627106.3627187

## 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 [57]. 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 [105] to password-based authentication [142]. 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) [151] or a distributed one [57, 91, 111, 118]. 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 [111] or a blockchain [118]). Without TTPs, most applications can mitigate forking attacks based on rollbacks by means of hardware-based monotonic counters [142]. 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 [131]. Previous work has shown that BI-SGX is vulnerable to forking attacks based on rollbacks [89]; the authors of [89] 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 [111] or NARRATOR [118] 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).
- **Prototype implementation & evaluation** We implemented a prototype of CloneBuster and evaluated it under realistic workloads (cf. Section 6). 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 [58].

## 2 BACKGROUND

### 2.1 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 [57].

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 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

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

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 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 [151] or distributed TTP [57, 91, 111, 118, 151]. For example, the authors of [57] show how to detect forking attacks if clients are mutually trusted—that is, clients themselves act as a distrusted TTP. Solutions like ROTE [111] or NARRATOR [118] 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 [118]. Once the cohort is forked, applications enclaves that use ROTE can be forked. ROTE [111] 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 3 (see Appendix). 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.

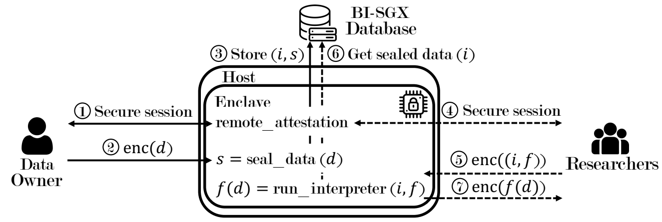



**Figure 1: Overview of the BI-SGX enclave and its interactions with Data Owners and Researchers**

- 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 [111].

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 the extended version of the paper [58].

## 3.3 Case Study: Cloning attack against BI-SGX

As a case-study, we show how to successfully mount a forking attack based on cloning against **BI-SGX** [131][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 [89]. Our attack against BI-SGX shows 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

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

**seal_data**

**Require:** Encrypted data $c_O$
$d$ = Decrypt $(c_O)$
$s$ = Seal $(d)$
**OCALL_Store** $(i, s)$

**run_interpreter**

**Require:** Encrypted request $c_R$
**Ensure:** Encrypted result $c_{RES}$
$req$ = Decrypt $(c_R)$
$i, f$ = BISGX_main $(req)$
$s$ = **OCALL_DB_get** $(i)$
$d$ = Unseal $(s)$
$c_{RES}$ = Encrypt $(f(d))$

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

**seal_data**

**Require:** Encrypted data $c_O$
$d$ = Decrypt $(c_O)$
**Increment(MC)**
**Read(MC)**
$s$ = Seal $(d, \mathrm{MC})$
**OCALL_Store** $(i, s)$

**run_interpreter**

**Require:** Encrypted request $c_R$
**Ensure:** Encrypted result $c_{RES}$
$req$ = Decrypt $(c_R)$
$i, f$ = BISGX_main $(req)$
$s$ = **OCALL_DB_get** $(i)$
$(d, \mathrm{MC})$ = Unseal $(s)$
**if** $i$ == MC **then**
$c_{RES}$ = Encrypt $(f(d))$

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

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., [89], 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., [89] 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., [89]. 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

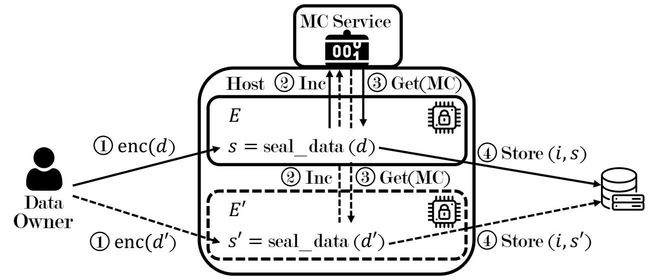



**Figure 4: Overview of a cloning attack against the fixed version of BI-SGX that uses monotonic counters.**

with the same index. We implemented the fix suggested by [89] 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)**. In a nutshell:

(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 [150] (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 6, 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.

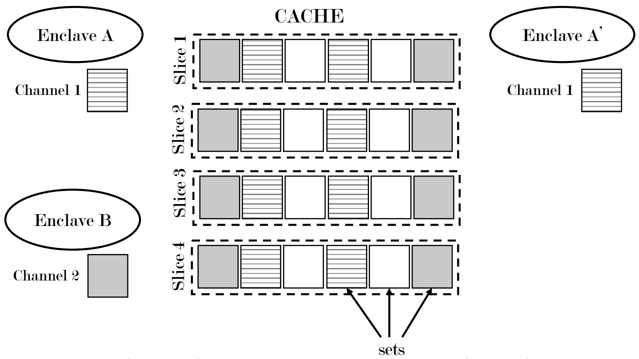



**Figure 5: Channels in CloneBuster. Each enclave uses a group of L3 cache sets to signal its presence and detect the presence of clones. Enclaves with the same binary (Enclaves A and A') use the same cache sets. Enclaves with different binaries (Enclaves A and B or A' and B) use different sets.**

# 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 3 (see Appendix), 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 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 [50, 60, 62, 63, 81, 100, 140, 143, 146], 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

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.

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

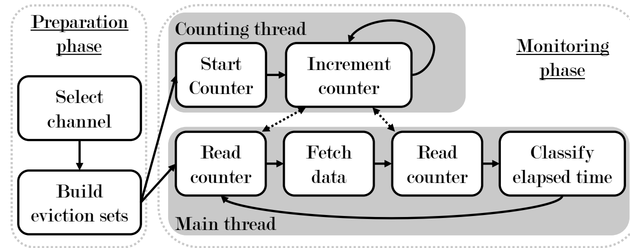



**Figure 6: Overview of CloneBuster.**

CloneBuster undergoes two phases of operation: a preparation phase and a monitoring phase. The preparation phase is used to define the "channel" to be used for signaling and detection. By channel, we refer to a specific group of cache sets, so that enclaves with the same (resp. different) binary will use the same (resp. a different) channel (cf. Figure 5).

Once the channel has been defined, CloneBuster builds the eviction sets required to communicate over such (cache-based) channel. During the monitoring phase, CloneBuster fills the cache sets of its channel with its own data, and continuously measures the time to access such data, in order to detect if it is still cached (cache hit) or if it has been evicted (cache miss). Note that clones will use the same channel (i.e., the same group of cache sets), removing each other's data. The resulting sequence of cache hits and misses is then fed to a classifier whose role is to distinguish whether clones are running on the same host based on the input sequence.

From an architectural point of view, CloneBuster relies on two threads. The main thread measures access time to the cache and runs the classifier in order to detect clones. Recall that SGX 1.0 enclaves have no access to high precision timers API (e.g., *rdtsc* and *rdtscp*). Thus, we leverage a second thread that implements a timer by continuously increasing a runtime variable [107, 137]. Figure 6 summarizes the main execution steps of CloneBuster. We note that SGX 2.0 allows enclaves to access *rdtsc*, so CloneBuster could work without the second thread on platforms where SGX 2.0 is available.

Notice that we do not define the specific enclave behavior in case the main thread detects a clone or if a clone raises an alarm, and leave the selection of a suitable choice to the enclave developer. However, it is reasonable to anticipate that the enclave would halt its execution and notify the owner in such cases.

Notice that the cache-based channel used by CloneBuster is shared with other applications and a potentially malicious OS. That is, any other process may intentionally pollute the channel of an enclave that uses CloneBuster. In case the channel is polluted, CloneBuster experiences a series of cache misses as if a clone were running on the same platform. Hence CloneBuster detects a clone and raises an alarm (e.g., stops the execution of the enclave). We treat this as a DoS attack and consider DoS attacks as out of scope.

In the following, we provide details on the preparation phase (channel selection and eviction sets) and the monitoring phase.

## 4.3 Phase 1: Preparation Phase

*4.3.1 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 the extended version of the paper [58].

Therefore, CloneBuster defines a channel as a group of 16 cache sets, in principle allowing for up to 64 concurrent channels. In the extended version of the paper [58],i 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 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.

*4.3.2 Building Eviction Sets.* In order to build eviction sets, the enclave must be aware of the specs of the CPU where it is deployed. This includes the number of slices, the number of sets per slice, and the number of ways per set. Such information must be hardcoded in the enclave. Alternatively, the enclave owner can pass such information to the enclave after the enclave has been deployed and the owner has attested it.

Popular techniques to build eviction sets from within an enclave [137] 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 [87]. Due to lack of space, we show in the extended version of the paper [58] that a malicious OS may evade detection if evictions sets are built relying on the assumption that enclave memory is contiguous. In particular, we show (using a SAT solver) that the OS can assign virtual memory to two instances of the same enclave so that they monitor different channels—effectively bypassing CloneBuster.

We leverage the technique of [87] to group data whose physical addresses share the last 20 bits and then regroup that data into

**Algorithm 1** Building the eviction sets in CloneBuster

```
Require: Memory byte array memArr[24MB];
Ensure: evictionSets[16][SLICES]
 1: spoilerArr[256][LIM] ← {}                         ▷ LIM depends on memArr size
 2: cacheGroups[16][16 * LIM] ← {}
 3: evictionSets[16][SLICES * WAYS] ← {}
 4: for i = 0 to 256 do
 5:     cont ← 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 ← 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][:] = reduce(cacheGroups[i][:])
```

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 rearranged 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 [86, 158]. If, on the contrary, the number of cores is a power of two, the aforementioned slice selection function [114] 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., [108, 120], 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 [108].

## 4.4 Phase 2: Monitoring

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 [137], we leverage a counting thread to measure access time: we fetch the value of the 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 main 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.

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., the expected time for a hit or the one for a miss. 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.

# 5 SECURITY ANALYSIS

**Knowledge of CPU specifications:** Note that CloneBuster requires the specifications of the processors where it is to be deployed. In particular, CloneBuster requires information on the cache, so to build eviction sets. Naturally, a malicious cloud provider may not faithfully report the CPU model where the enclave is going to be deployed. However, we believe that a rational cloud provider has no incentive to provide fabricated information on its CPUs. This is because if the malicious behavior of the cloud is exposed, its reputation may be severely affected. In a nutshell, CloneBuster is not designed to counter a malicious cloud provider, but rather an adversary that compromises the OS on the cloud machines.

**Changing channel assignment:** 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 [79], which in turn is a wrapper of MiniSAT [73]) 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 the extended version of the paper [58].

**Side-stepping the enclave:** 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 [150]. 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). Notice that other interruption strategies, besides single-stepping the enclave, could be used by the adversary. For instance, the adversary might try to infrequently interrupt either clones in an attempt to prevent detection. While such attacks could result in a false positive (raising an alarm by CloneBuster), it remains unclear whether CloneBuster can comprehensively detect all such attack strategies.

**Polluting the channel:** 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. We confirm this by experiments in Section 6.

**Slowing down threads:** 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 main thread measures access times to its eviction set. The OS can achieve this by scheduling the counting thread on a core along with other applications. This strategy would slow down the counting thread and result in anomalous readings by the main thread. Here, the main thread reads the counter and computes an elapsed time value that does not match the elapsed time of a cache miss nor it matched the elapsed time of a cache miss. The current version of CloneBuster does not address such attack. However, we believe that it could be addressed by having the main thread raising an alarm every time it detects an anomalous reading of the counter. We have empirically verified this by scheduling threads on the same core where the counting thread was running and the main thread witnessed no increases of the counter variable. Similarly, if the adversary slows down the main thread, the corresponding AEX could be detected by monitoring the SSA area as done in previous work [119]; once the thread resumes and detects the asynchronous exit, it could raise an alarm.

**Modifying core frequency:** 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 main 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 [82, 134]. 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 [126, 141]. We have empirically verified this in our platform. Even assuming future processors with SGX and per-core frequency scaling [85], some time elapses between the instant when the OS makes a frequency change request until this change is effective. As reported in [82, 134], this time interval amounts to roughly 500 $\mu$s; in contrast a cache miss only takes around 0.15 $\mu$s. Thus, adding a periodic re-calibration phase where the main thread measures the time of a cache miss, may prevent the OS from scaling the frequency. In particular, if the re-calibration phase occurs every 500 $\mu$s, 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 could be spotted with periodic re-calibration.

**Changing memory mapping:** A malicious OS may change the physical to virtual mapping by leveraging its ability to control some of the bits of an address that determine the cache set (bits 12-15). We note that the enclave fixes bits 6-11 and monitors all sets corresponding to all configurations of the remaining 4 bits. As an example, fixed bits 6-11 as 010101, then CloneBuster monitors the sets given as XXXX010101 where XXXX ranges from 0000 to 1111. In case the OS changes the mapping between a virtual address and a physical address (e.g., by swapping pages using the EWB instruction) an address would move from one of the sets monitored by CloneBuster to another set that is also monitored by CloneBuster. In this case, CloneBuster may end up polluting its own cache sets. If the number of addresses that underwent a change set is high, CloneBuster would mistakenly detect a clone and raise an alarm. This is another case of false positive and, as mentioned before, false positives are not in the attacker's best interest. Changes to the mapping between physical and virtual addresses may also be carried differently, e.g., by adding and removing pages via EDMM (available for SGX 2.0). While we could not verify this strategy on SGX 2.0, we speculate, however, that such changes made to the page mappings using EDMM are likely to trigger a notification before becoming effective, which, in turn, can be detected by CloneBuster (see [115] page 3, Section 3.1 for more details).

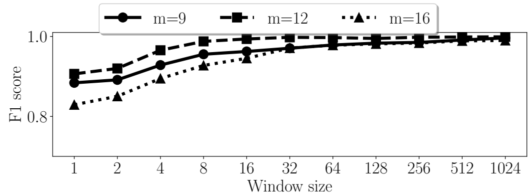
# 6 IMPLEMENTATION AND EVALUATION

## 6.1 Implementation Setup

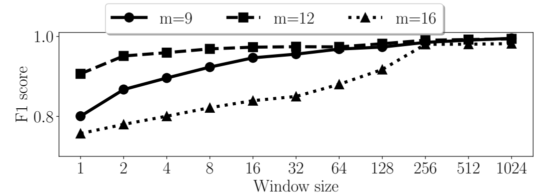
We implemented a prototype of CloneBuster, including the code for the creation of the eviction sets (and some tests to ensure they have been properly built) and the counting thread that serves as a clock. Our implementation accounts for approximately 800 LoC.

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 $w$.

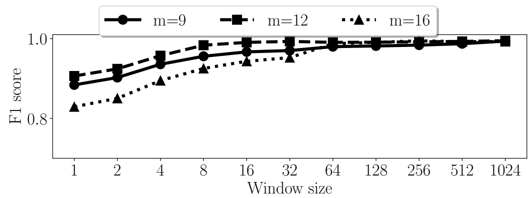
We evaluate performance in an ideal scenario where no other application apart from the enclave (and possibly its clone) is running, as well as in a more realistic scenario where background processes—taken from the Phoronix benchmark suite [101]—are running on the host at the same time. In scenarios featuring 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.



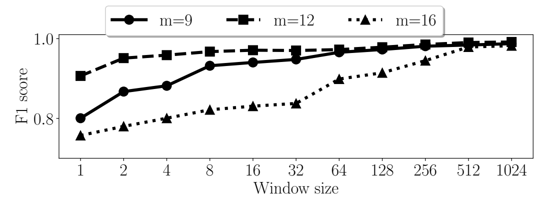

**(a) Threshold**



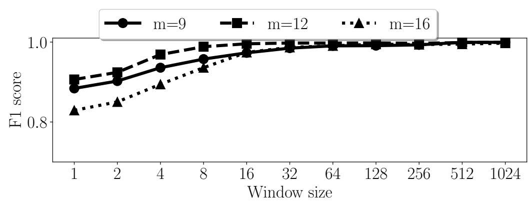

**(b) Threshold (x265)**



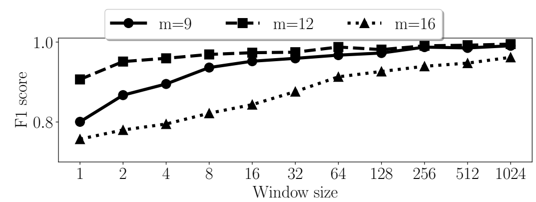

**(c) Gaussian Naive Bayes**




**(d) Gaussian Naive Bayes (x265)**




**(e) Neural Network**




**(f) Neural Network (x265)**

**Figure 7: 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.**

## 6.2 Evaluation Results

We assess the performance of CloneBuster by means of F1 score. We additionally report in Tables 1 and 2 (see Appendix) the associated false positive and negative rates for each experiment. 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 [121] 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 7 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.

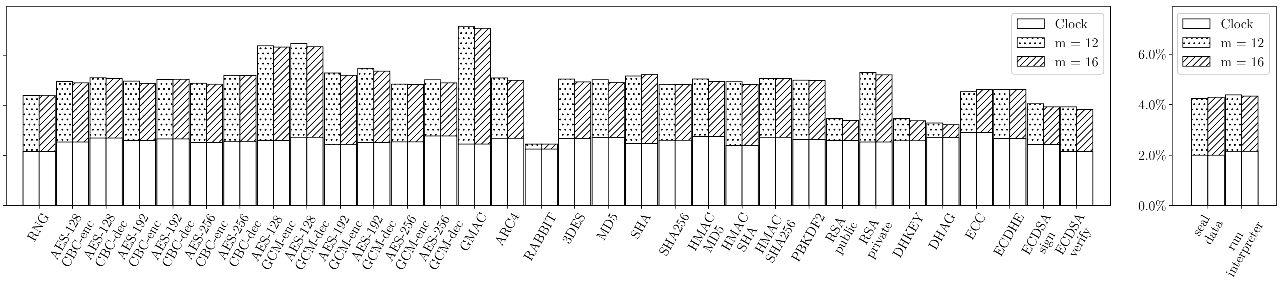
The comparison between the plots on the left side of Figure 7 (with no background process) and the ones on the right side of the same figure (with `x265 video encoder` running in parallel) allows us to assess the impact of background processes on the performance.

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 (and its associated false positive/negative rates).

In the extended version of the paper [58], we provide additional results with alternative detection algorithms and background applications of the Phoronix benchmark suite.

**Impact of observation window size $w$.** Figure 7 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




**(a) WolfSSL** **(b) BI-SGX**

**Figure 8: 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 9: 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.**

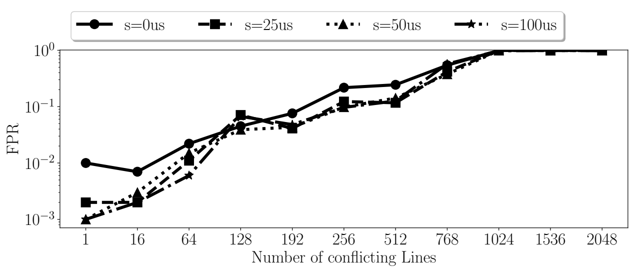



**Figure 10: False positive rate when a malicious OS injects noise by placing into the cache based covert channel different amount of cache lines. $s$ refers to the waiting interval between injections of noise**

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 (cf. Table 2). On the other hand, performance difference between $m = 9$ and $m = 12$, depends on the process running in the background. For instance, when $w = 256$, as shown in Tables 1 and 2, the number of false positives is generally higher for $m = 12$ and the number of false negatives is generally higher for $m = 9$. Since we consider false negatives more damaging than false positives, we opt for $m = 12$.

**Impact of malicious noise on detection.** As discussed in Section 5, a malicious OS might artificially add noise to the channel. We have tested such scenario with the following experiment. We run CloneBuster (with the threshold-based classifier, $w = 64$ and $m = 12$) while increasing both the number of lines in the channel polluted by the OS, as well as the frequency with which the OS pollutes those lines. The number of lines polluted by the OS ranged from 1 to 2048 (i.e., from one to all lines of the cache sets monitored by CloneBuster); the OS injected noise in intervals of 0, 25, 50, and 100 $\mu$s. Figure 10 shows the impact of such strategy on the false positive rate. Our results show that if the adversary can pollute more than 768 cache lines, CloneBuster always results in a false positive. Conversely, when the adversary pollutes less than 192 cache lines, the resulting FPR is very low. On the other hand, our experiments show that this strategy does not impact the false negative rate of CloneBuster (it consistently remains between 0 and 0.009).

**Performance overhead for WolfSSL.** We use applications of WolfSSL [155]—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 vanilla version as baseline and compare its throughput with the one of the same application when enhanced with CloneBuster.

Figure 8 (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 main 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 [131] when CloneBuster is used to detect attacks. Figure 8 (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 9, 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 9 shows that even with background noise, the F1 score reaches 0.999 for $w \geq 64$, with a false positive rate of 0.0015 and a false negative rate of 0.0004.

## 7 CONCLUDING REMARKS

In this work, we addressed the problem of forking attacks against Intel SGX by cloning the victim enclave. We 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. 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.

## ACKNOWLEDGMENTS

This work is partly funded by the Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under Germany's Excellence Strategy - EXC 2092 CASA - 390781972, and the European Union's Horizon 2020 research and innovation programme (SPATIAL, Grant Agreement No. 101021808). Views and opinions expressed are however those of the authors only and do not necessarily reflect those of the European Union. Neither the European Union nor the granting authority can be held responsible for them.

## 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] 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
[51] 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
[52] 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.
[53] 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.
[54] 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
[55] Mic Bowman, Andrea Miele, Michael Steiner, and Bruno Vavala. 2018. Private data objects: an overview. *arXiv preprint arXiv:1807.05686* (2018).
[56] 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).
[57] 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.
[58] Samira Briongos, Ghassan Karame, Claudio Soriente, and Annika Wilde. 2023. No Forking Way: Detecting Cloning Attacks on Intel SGX Applications. arXiv:2310.03002 [cs.CR]
[59] Somnath Chakrabarti, Brandon Baker, and Mona Vij. 2017. Intel SGX enabled key manager service with openstack barbican. *arXiv preprint arXiv:1712.07694* (2017).
[60] 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.
[61] 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
[62] 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.
[63] 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
[64] 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
[65] 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
[66] 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
[67] 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
[68] 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
[69] 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
[70] 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.
[71] 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
[72] 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
[73] Niklas Eén and Niklas Sörensson. 2004. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing*. Springer Berlin Heidelberg, Berlin, Heidelberg, 502–518.
[74] Saba Eskandarian and Matei Zaharia. 2017. Oblidb: Oblivious query processing for secure databases. *arXiv preprint arXiv:1710.00458* (2017).
[75] 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
[76] MobileCoin Foundation. 2019. MobileCoin. https://github.com/mobilecoinfoundation/mobilecoin.
[77] 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.
[78] 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
[79] giganetom et al. 2018. SATisPy. https://github.com/netom/satispy. Accessed: 2020-10-01.
[80] 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
[81] 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
[82] 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
[83] 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.
[84] Hyperledger. 2018. Hyperledger Fabric Private Chaincode. https://github.com/hyperledger/fabric-private-chaincode.
[85] Intel(R). 2021. Intel® Xeon® Scalable Processors. https://www.intel.com/content/www/us/en/products/details/processors/xeon/scalable.html.
[86] 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.
[87] 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
[88] 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)*.
[89] 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.
[90] Prasad Koshy Jose. 2020. Confidential Computing of Machine Learning using Intel SGX. https://github.com/prasadkjose/confidential-ml-sgx.
[91] 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.

[92] 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

[93] 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

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

[95] 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.

[96] 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

[97] 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

[98] 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

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

[100] 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.

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

[102] 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

[103] 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.

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

[105] 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).

[106] 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

[107] 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

[108] 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

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

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

[111] 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.

[112] 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.

[113] 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.

[114] 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.

[115] Frank McKeen, Ilya Alexandrovich, Ittai Anati, Dror Caspi, Simon Johnson, Rebekah Leslie-Hurd, and Carlos V. Rozas. 2016. Intel Software Guard Extensions (Intel® SGX) Support for Dynamic Memory Management Inside an Enclave. In *HASP@ISCA*. 10:1–10:9.

[116] 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

[117] 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

[118] 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.

[119] Oleksii Oleksenko, Bohdan Trach, Robert Krahn, Mark Silberstein, and Christof Fetzer. 2018. Varys: Protecting SGX Enclaves from Practical Side-Channel Attacks. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*. USENIX Association, Boston, MA, 227–240. https://www.usenix.org/conference/atc18/presentation/oleksenko

[120] 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

[121] 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.

[122] 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

[123] 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

[124] 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.

[125] 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

[126] 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

[127] 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

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

[129] 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

[130] 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
[131] Aoi Sakurai. 2019. BI-SGX: Secure Cloud Computation. https://github.com/hello31337/BI-SGX. Accessed: 2023-01-16.
[132] 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
[133] Sajin Sasy and Ian Goldberg. 2019. ConsenSGX: Scaling Anonymous Communications Networks with Trusted Execution Environments. *Proc. Priv. Enhancing Technol.* 2019, 3 (2019), 331–349.
[134] 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
[135] 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
[136] 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.
[137] 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.
[138] 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
[139] 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
[140] 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
[141] 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
[142] Raoul Strackx and Frank Piessens. 2016. Ariadne: A Minimal Approach to State Continuity. In *25th USENIX Security Symposium*. 875–892.
[143] 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).
[144] 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
[145] 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
[146] 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.
[147] Florian Tramer and Dan Boneh. 2018. Slalom: Fast, verifiable and private execution of neural networks in trusted hardware. *arXiv preprint arXiv:1806.03287* (2018).
[148] Florian Tramèr and Dan Boneh. 2018. SLALOM. https://github.com/ftramer/slalom.
[149] 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
[150] 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
[151] 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.
[152] 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.
[153] 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.
[154] 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
[155] wolfSSL. 2017. wolfSSL Linux Enclave Example. https://github.com/wolfSSL/wolfssl-examples/tree/master/SGX_Linux. Accessed: 2020-21-04.
[156] 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.
[157] 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
[158] 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
[159] Hang Yin, Shunfan Zhou, and Jun Jiang. 2019. Phala network: A confidential smart contract network based on polkadot.
[160] 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
[161] Fan Zhang, Ethan Cecchetti, Kyle Croman, Ari Juels, and Elaine Shi. 2016. Town Crier: An Authenticated Data Feed for Smart Contracts. In *Proceedings of 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
[162] 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.


# APPENDIX

We show the false negative/positive rates in Tables 1 and 2. In Table 3, we summarized our analysis of SGX applications. Here, 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 the extended version of the paper [58]. Proof of Luck (⋆) 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.

| | | 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 | 0.113 | 0.087 | 0.051 | 0.035 | 0.023 | 0.012 | 0.102 | 0.078 | 0.039 | 0.021 | 0.022 | 0.004 | 0.103 | 0.068 | 0.027 | 0.009 | 0.007 | 0.001 |
| | m=12 | 0.097 | 0.024 | 0.002 | 0.001 | 0.001 | 0.000 | 0.098 | 0.045 | 0.009 | 0.009 | 0.005 | 0.004 | 0.098 | 0.030 | 0.004 | 0.003 | 0.003 | 0.000 |
| | m=16 | 0.158 | 0.107 | 0.064 | 0.021 | 0.012 | 0.001 | 0.113 | 0.103 | 0.021 | 0.011 | 0.003 | 0.006 | 0.114 | 0.102 | 0.021 | 0.008 | 0.002 | 0.001 |
| x265 | m=9 | 0.189 | 0.108 | 0.048 | 0.007 | 0.002 | 0.001 | 0.182 | 0.119 | 0.061 | 0.034 | 0.019 | 0.015 | 0.189 | 0.108 | 0.049 | 0.038 | 0.014 | 0.009 |
| | m=12 | 0.082 | 0.043 | 0.021 | 0.021 | 0.003 | 0.002 | 0.091 | 0.045 | 0.027 | 0.021 | 0.012 | 0.003 | 0.092 | 0.039 | 0.016 | 0.010 | 0.006 | 0.004 |
| | m=16 | 0.198 | 0.156 | 0.153 | 0.102 | 0.012 | 0.009 | 0.210 | 0.178 | 0.134 | 0.098 | 0.041 | 0.009 | 0.211 | 0.189 | 0.123 | 0.085 | 0.054 | 0.023 |
| sql | m=9 | 0.208 | 0.098 | 0.053 | 0.041 | 0.010 | 0.005 | 0.212 | 0.102 | 0.056 | 0.021 | 0.011 | 0.008 | 0.199 | 0.191 | 0.042 | 0.012 | 0.007 | 0.001 |
| | m=12 | 0.043 | 0.021 | 0.019 | 0.018 | 0.010 | 0.006 | 0.045 | 0.023 | 0.018 | 0.008 | 0.006 | 0.004 | 0.045 | 0.023 | 0.018 | 0.009 | 0.006 | 0.003 |
| | m=16 | 0.198 | 0.134 | 0.124 | 0.098 | 0.052 | 0.018 | 0.187 | 0.152 | 0.146 | 0.127 | 0.078 | 0.012 | 0.187 | 0.160 | 0.129 | 0.085 | 0.013 | 0.013 |
| opencv | m=9 | 0.223 | 0.102 | 0.072 | 0.065 | 0.043 | 0.021 | 0.161 | 0.126 | 0.087 | 0.081 | 0.047 | 0.030 | 0.161 | 0.125 | 0.086 | 0.075 | 0.039 | 0.027 |
| | m=12 | 0.091 | 0.065 | 0.058 | 0.045 | 0.032 | 0.012 | 0.090 | 0.062 | 0.054 | 0.042 | 0.031 | 0.011 | 0.090 | 0.062 | 0.054 | 0.039 | 0.021 | 0.010 |
| | m=16 | 0.320 | 0.213 | 0.211 | 0.193 | 0.072 | 0.034 | 0.310 | 0.223 | 0.132 | 0.092 | 0.089 | 0.032 | 0.310 | 0.218 | 0.081 | 0.070 | 0.034 | 0.021 |
| gcc | m=9 | 0.163 | 0.124 | 0.073 | 0.043 | 0.032 | 0.028 | 0.159 | 0.099 | 0.050 | 0.032 | 0.017 | 0.017 | 0.159 | 0.099 | 0.056 | 0.024 | 0.018 | 0.017 |
| | m=12 | 0.068 | 0.031 | 0.019 | 0.009 | 0.006 | 0.005 | 0.069 | 0.026 | 0.018 | 0.013 | 0.013 | 0.007 | 0.069 | 0.026 | 0.017 | 0.012 | 0.013 | 0.010 |
| | m=16 | 0.190 | 0.176 | 0.132 | 0.102 | 0.087 | 0.054 | 0.192 | 0.162 | 0.112 | 0.096 | 0.087 | 0.017 | 0.191 | 0.161 | 0.103 | 0.087 | 0.068 | 0.036 |
| cloud | m=9 | 0.067 | 0.059 | 0.031 | 0.023 | 0.017 | 0.010 | 0.079 | 0.045 | 0.031 | 0.032 | 0.020 | 0.010 | 0.081 | 0.043 | 0.025 | 0.021 | 0.009 | 0.006 |
| | m=12 | 0.100 | 0.071 | 0.047 | 0.018 | 0.007 | 0.003 | 0.091 | 0.040 | 0.032 | 0.012 | 0.012 | 0.002 | 0.083 | 0.032 | 0.023 | 0.009 | 0.004 | 0.003 |
| | m=16 | 0.142 | 0.090 | 0.061 | 0.011 | 0.009 | 0.009 | 0.109 | 0.089 | 0.056 | 0.043 | 0.021 | 0.013 | 0.102 | 0.056 | 0.049 | 0.037 | 0.013 | 0.006 |

**Table 1: False negative rates for various 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.**

| | | 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 | 0.120 | 0.055 | 0.024 | 0.008 | 0.007 | 0.004 | 0.134 | 0.048 | 0.027 | 0.019 | 0.010 | 0.008 | 0.132 | 0.059 | 0.027 | 0.009 | 0.007 | 0.001 |
| | m=12 | 0.090 | 0.047 | 0.010 | 0.005 | 0.003 | 0.002 | 0.089 | 0.041 | 0.009 | 0.009 | 0.011 | 0.006 | 0.089 | 0.032 | 0.004 | 0.003 | 0.003 | 0.002 |
| | m=16 | 0.189 | 0.103 | 0.045 | 0.023 | 0.020 | 0.019 | 0.253 | 0.107 | 0.019 | 0.015 | 0.007 | 0.008 | 0.252 | 0.111 | 0.027 | 0.010 | 0.010 | 0.003 |
| x265 | m=9 | 0.214 | 0.099 | 0.059 | 0.006 | 0.004 | 0.001 | 0.224 | 0.117 | 0.059 | 0.032 | 0.019 | 0.015 | 0.214 | 0.101 | 0.047 | 0.028 | 0.012 | 0.009 |
| | m=12 | 0.106 | 0.039 | 0.033 | 0.031 | 0.015 | 0.010 | 0.095 | 0.037 | 0.031 | 0.033 | 0.018 | 0.013 | 0.094 | 0.043 | 0.039 | 0.016 | 0.012 | 0.006 |
| | m=16 | 0.317 | 0.263 | 0.172 | 0.143 | 0.028 | 0.027 | 0.297 | 0.230 | 0.218 | 0.105 | 0.071 | 0.027 | 0.296 | 0.229 | 0.201 | 0.089 | 0.067 | 0.054 |
| sql | m=9 | 0.055 | 0.052 | 0.038 | 0.022 | 0.010 | 0.005 | 0.085 | 0.056 | 0.027 | 0.003 | 0.005 | 0.002 | 0.071 | 0.073 | 0.038 | 0.004 | 0.003 | 0.003 |
| | m=12 | 0.068 | 0.033 | 0.025 | 0.036 | 0.016 | 0.010 | 0.066 | 0.031 | 0.028 | 0.016 | 0.014 | 0.006 | 0.041 | 0.031 | 0.026 | 0.021 | 0.014 | 0.009 |
| | m=16 | 0.173 | 0.181 | 0.176 | 0.098 | 0.037 | 0.022 | 0.189 | 0.169 | 0.167 | 0.122 | 0.054 | 0.026 | 0.189 | 0.148 | 0.157 | 0.089 | 0.069 | 0.038 |
| opencv | m=9 | 0.065 | 0.050 | 0.044 | 0.040 | 0.026 | 0.005 | 0.124 | 0.092 | 0.063 | 0.053 | 0.043 | 0.028 | 0.121 | 0.096 | 0.043 | 0.029 | 0.018 | 0.015 |
| | m=12 | 0.104 | 0.076 | 0.069 | 0.039 | 0.034 | 0.020 | 0.106 | 0.079 | 0.075 | 0.046 | 0.041 | 0.021 | 0.106 | 0.079 | 0.071 | 0.027 | 0.033 | 0.016 |
| | m=16 | 0.350 | 0.244 | 0.012 | 0.021 | 0.083 | 0.040 | 0.370 | 0.228 | 0.132 | 0.160 | 0.085 | 0.063 | 0.370 | 0.236 | 0.077 | 0.066 | 0.055 | 0.037 |
| gcc | m=9 | 0.132 | 0.051 | 0.016 | 0.033 | 0.024 | 0.018 | 0.138 | 0.071 | 0.050 | 0.012 | 0.017 | 0.017 | 0.138 | 0.071 | 0.027 | 0.004 | 0.016 | 0.009 |
| | m=12 | 0.070 | 0.023 | 0.021 | 0.023 | 0.024 | 0.013 | 0.069 | 0.034 | 0.030 | 0.067 | 0.021 | 0.009 | 0.069 | 0.028 | 0.023 | 0.026 | 0.021 | 0.012 |
| | m=16 | 0.192 | 0.143 | 0.118 | 0.080 | 0.083 | 0.062 | 0.190 | 0.164 | 0.132 | 0.100 | 0.061 | 0.013 | 0.191 | 0.163 | 0.107 | 0.078 | 0.068 | 0.036 |
| cloud | m=9 | 0.067 | 0.024 | 0.017 | 0.015 | 0.007 | 0.010 | 0.053 | 0.020 | 0.001 | 0.024 | 0.008 | 0.006 | 0.051 | 0.022 | 0.009 | 0.011 | 0.005 | 0.004 |
| | m=12 | 0.087 | 0.062 | 0.047 | 0.022 | 0.013 | 0.011 | 0.098 | 0.080 | 0.055 | 0.016 | 0.014 | 0.016 | 0.107 | 0.089 | 0.067 | 0.015 | 0.012 | 0.013 |
| | m=16 | 0.147 | 0.077 | 0.070 | 0.027 | 0.017 | 0.015 | 0.191 | 0.078 | 0.075 | 0.066 | 0.035 | 0.031 | 0.200 | 0.106 | 0.081 | 0.069 | 0.027 | 0.014 |

**Table 2: False positive rates for various 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.**

| Project | Source code available | Vulnerable to | |
|---|---|---|---|
| | | Rollback | Cloning |
| **Encrypted Databases and Key-value Stores** | | | |
| **Aria** [156] [p] | No | N/A | Yes (A) |
| **Avocado** [38, 52] [a] | Yes | N/A | Yes (A) |
| **Enclage** [144] [p] | No | N/A | Yes (A) |
| **EnclaveCache** [61] [p] | No | Yes | Yes (B) |
| **EnclaveDB** [125] [p] | No | No (MC) | No (TTP) |
| **HardIDX** [77] [p] | No | Yes | N/A |
| **NeXUS** [7, 69] [p] | Yes | No (MC) | Yes (B) |
| **ObliDB** [8, 74] [p] | Yes | N/A | Yes (A) |
| **PESOS** [96] [p] | No | N/A | N/A |
| **SeGShare** [78] [p] | No | No (MC) | N/A |
| **ShieldStore** [22, 93] [ap] | Yes | No (MC) | Yes (B) |
| **SPEICHER** [1, 53] [a] | Yes | No (MC) | No (TTP) |
| **STANlite** [35, 132] [ap] | Yes | N/A | Yes (A) |
| **StealthDB** [15, 152] [a] | Yes | Yes | Yes (B) |
| **Applications** | | | |
| **BI-SGX** [16] [a] | Yes | No (MC) | Yes (B) |
| **CACIC** [49, 145] [a] | Yes | Yes | Yes (B) |
| **DEBE** [44, 157] [a] | Yes | N/A | N/A |
| **HySec-Flow** [39, 154] [a] | Yes | N/A | N/A |
| **PrivaTube** [66] [p] | No | N/A | Yes (C) |
| **REX** [47, 68] [a] | Yes | N/A | N/A |
| **SGXDedup** [37, 128] [a] | Yes | N/A | N/A |
| **Signal CDS** [11, 110] [a] | Yes | N/A | N/A |
| **SkSES** [23, 95] [a] | No | N/A | N/A |
| **SMac** [34, 70] [a] | Yes | N/A | N/A |
| **TresorSGX** [5, 129] [a] | Yes | N/A | N/A |
| **Key + Password Management** | | | |
| **DelegaTEE** [112] [p] | No | No (MC) | N/A |
| **FeIDo** [45, 135] [a] | Yes | N/A | N/A |
| **Keys in Clouds** [17, 98] [a] | Yes | No (MC) | N/A |
| **SafeKeeper** [21, 97] [a] | Yes | No (MC) | N/A |
| **SGX-KMS** [12, 59] [a] | Yes | Yes | Yes (B) |
| **Private Search** | | | |
| **BISEN** [25, 75] [a] | Yes | N/A | N/A |
| **DeSearch** [40, 103] [a] | Yes | N/A | N/A |
| **Maiden** [33, 153] [a] | Yes | N/A | N/A |
| **POSUP** [20, 83] [a] | Yes | N/A | N/A |
| **QShield** [30, 64] [a] | Yes | N/A | N/A |
| **Snoopy** [42, 67] [a] | Yes | No (MC) | N/A |

| Project | Source code available | Vulnerable to | |
|---|---|---|---|
| | | Rollback | Cloning |
| **X-Search** [36, 117] [ap] | Yes | N/A | Yes (C) |
| **Blockchains** | | | |
| **BITE** [113] [p] | No | No (MC) | N/A |
| **BLOXY** [130] [p] | No | N/A | N/A |
| **Ekiden** [6, 65] [a] | Yes | No (TTP) | No (TTP) |
| **Hybrids on Steroids** [54] [p] | No | No (MC+TTP) | No (TTP) |
| **MobileCoin** [41, 76] [a] | Yes | No (TTP) | No (TTP) |
| **Oasis** [19] [a] | Yes | No (TTP) | No (TTP) |
| **Obscuro** [9, 149] [a] | Yes | N/A | N/A |
| **Phala Network** [27, 159] [a] | Yes | No (TTP) | No (TTP) |
| **Private Chaincode** [56, 84] [a] | Yes | N/A | N/A |
| **Private Data Objects** [55, 99] [a] | Yes | No (TTP) | No (TTP) |
| **Proof of Luck** [2, 116] [a] | Yes | N/A | No (★) |
| **Teechain** [104, 106] [ap] | Yes | No (MC) | N/A |
| **Town Crier** [4, 161] [a] | Yes | No (TTP) | No (TTP) |
| **Troxy** [54] [p] | No | N/A | No (TTP) |
| **Twilight** [46, 71] [a] | Yes | N/A | N/A |
| **Machine Learning** | | | |
| **Confidential ML** [90] [a] | Yes | N/A | N/A |
| **DP-GBDT** [109] [a] | Yes | No (MC) | N/A |
| **Plinius** [29, 160] [a] | Yes | No (MC) | N/A |
| **secureTF** [127] [p] | No | N/A | N/A |
| **Secure XGBoost** [31, 102] [a] | Yes | N/A | N/A |
| **Slalom** [147, 148] [ap] | Yes | N/A | N/A |
| **SOTER** [43, 138] [a] | Yes | N/A | N/A |
| **Network** | | | |
| **ConsenSGX** [26, 133] [a] | No | N/A | N/A |
| **CYCLOSA** [122] [p] | No | N/A | N/A |
| **ENDBOX** [80] [ap] | No | N/A | N/A |
| **LightBox** [18, 72] [ap] | Yes | N/A | N/A |
| **SENG** [32, 136] [a] | Yes | N/A | N/A |
| **SGX CBR** [123] [p] | No | N/A | N/A |
| **SGX-Tor** [14, 92] [ap] | Yes | Yes | N/A |
| **TEE V2V** [48, 88] [a] | No | N/A | N/A |
| **MACSec** [94] [a] | Yes | No (MC) | N/A |
| **S-NFV** [139] [p] | No | N/A | N/A |
| **SafeBricks** [3, 124] [ap] | Yes | N/A | N/A |
| **SELIS-PubSub** [28, 51] [p] | Yes | N/A | N/A |
| **Data Analytics** | | | |
| **Opaque** [10, 162] [a] | Yes | No (TTP) | No (TTP) |

**Table 3: Summary of our analysis of SGX applications. We analysed SGX applications listed in [13] (superscript $p$ next to the citation) and listed in [24] (superscript $a$ next to the citation).**