Software-Based Protection against Changeware

Sebastian Banescu
Alexander Pretschner
Technische Universität München, Germany
{banescu,pretschn}@cs.tum.edu

Dominic Battré, Stéfano Cazzulani
Robert Shield, Greg Thompson
Google Inc.
{battre,stefanoc,robertshield,grt}@google.com

ABSTRACT

We call *changeware* software that surreptitiously modifies resources of software applications, e.g., configuration files. Changeware is developed by malicious entities which gain profit if their changeware is executed by large numbers of end-users of the targeted software. *Browser hijacking* malware is one popular example that aims at changing web-browser settings such as the default search engine or the home page. Changeware tends to provoke end-user dissatisfaction with the target application, e.g. due to repeated failure of persisting the desired configuration. We describe a solution to counter changeware, to be employed by vendors of software targeted by changeware. It combines several protection mechanisms: white-box cryptography to hide a cryptographic key, software diversity to counter automated key retrieval attacks, and run-time process memory integrity checking to avoid illegitimate calls of the developed API.

Categories and Subject Descriptors

K.6.5 [**Security and Protection**]: Invasive software

Keywords

Software protection; Malware defense; Integrity protection; White-box cryptography; Obfuscation; Software Diversity

1. INTRODUCTION

Some malware surreptitiously attacks the integrity of specific software assets, not their confidentiality or availability. An attack is successful if malware is able to automatically change specific assets of a target software in accordance with the attacker's wishes, when executed on a large number of victims' devices. We call this type of malware *changeware* throughout the remainder of this paper. The typical changeware attack scenario involves three types of participants: (1) The *software vendor* and distributor of a software called X that consists of both read-only binaries (signed by the OS vendor) and editable assets (e.g., configuration files) that must be modifiable by X; (2) *end-users* of X (victims of changeware) who we assume numerous (thousands to hundreds of millions) and who download X from the Internet; and (3) *changeware developers* who gain a monetary or other advantage proportional to the number of remote systems of legitimate X end-users they successfully attack.

The goal of the changeware developer is to attack a large number of legitimate end-users of X by: (1) creating an automated attack in the form of a computer program (i.e., changeware) and (2) tricking end-users into executing it. Changeware is less complex than other types of malware, because it does not contain exploits to gain root privileges. This is because it does not need such privileges for a successful attack. Therefore, we assume that changeware does not have root privileges during its execution.

If an end-user executes changeware, all *editable* software resources (assets) associated with X, i.e., not protected by the underlying operating system (OS) code signature verification mechanism, are subject to unsolicited modification because changeware has the same privileges as the currently authenticated OS user. Note that modification of non-editable software assets such as application binaries signed by the OS vendor are detected and signaled to the end-user by the OS. However, applications generally also need editable resources (e.g. configuration files), which are prone to unsolicited modification attacks. One popular example is browser-hijacking if X is a web-browser.

These attacks have become popular nowadays because of the possibility of bundling changeware with benign software into the same executable installer. The end-user (victim) is tricked into installing a seemingly legitimate software (i.e., digitally signed by a trusted vendor) which also installs changeware transparently for the victim. The high success-rate of this social-engineering step of the attack is appealing to changeware developers since it eliminates the task of bypassing network and operating-system security mechanisms (e.g. firewalls, authentication).

Unfortunately, changeware is not detected by most commercial anti-virus software, as a consequence of its seemingly legitimate behavior. Changeware modifies software assets belonging to the same OS user under whose privileges the attacked software is also running. Since access control in OSs like Microsoft Windows, Linux, Mac OS, etc. is user-centric and not application-centric, changeware has the right to edit assets that belong to other applications.

Changeware can also write the memory of any other process running under the same OS user privileges. Several OSs (see Section 3.4.1) even offer the possibility to start a thread

CODASPY'15, March 2–4, 2015, San Antonio, Texas, USA.
ACM 978-1-4503-3191-3/15/03.
http://dx.doi.org/10.1145/2699026.2699099 .

inside a process, which executes code that was previously written to (*injected into*) the process memory by changeware. To make things more difficult, on Windows OSs, code injection, in the form of dynamic-link library (DLL) injection, is also executed by benign software (e.g. anti-virus applications) and therefore not considered malicious.

Leveraging a *trusted entity* such as additional hardware (e.g. the Trusted Platform Module[30], smart-cards), to protect the integrity of software assets is possible. Such trusted entities contain a hardware protected secret-key which can be used to compute a message authentication code of software asset values. However, such a trusted entity implies additional costs for end-users, usage or setup inconvenience and possibly privacy concerns. Another approach for defending against changeware would be to re-authenticate the end-user via the OS password prompt, whenever changes to software assets occur. This way, writing to software assets is done by the OS kernel, which only performs the changes if the user confirms the changes via re-authentication. However, this would negatively impact user experience and may become too tedious for practical use. Therefore, a solution which is transparent to the end-user is preferable.

This paper addresses the problem of how to protect editable software assets and makes the following contributions:

1. A novel software-only solution against changeware that (a) can be directly employed by the targeted software vendor; (b) is transparent to the end-user; and (c) does not require communication with trusted entities. The solution leverages three distinct protection mechanisms: *white-box cryptography* [5], *software diversity* [13], and *run-time process memory invariant checking*.
2. A set of data obfuscation transformations at the level of source code. These hide the position of white-box cryptographic ciphers inside the data segment of an executable binary.
3. An inter-process run-time checking mechanism via code injection. This verifies that synchronous function calls are the ones intended by the software vendor. Verification is performed against a white-list of fixed-size precomputed OS version-dependent signatures of the call-stack of the calling thread and its associated code.
4. An implementation and evaluation of our solution in the form of a case-study on protecting the Chromium web-browser user preference against browser hijacking.

The remainder of this paper is structured as follows. After briefly presenting related work (Section 2), we describe our solution in Section 3, the core of this paper. We evaluate and discuss guarantees and limitations in Sections 4 and 5. Finally, conclusions and directions for future work are presented in Section 6.

2. RELATED WORK

Since our candidate solution encompasses white-box cryptographic primitives, software diversity and run-time integrity checking, we structure related work accordingly.

2.1 White-Box Cryptography

White-box cryptography (WBC) was pioneered by Chow *et al.* [6, 5], who proposed the first white-box DES, respectively white-box AES (WB-AES) ciphers in 2002. The goal of white-box cryptography is the secure storage of secret keys (used by cryptographic ciphers), in software, without hardware keys or trusted entities. Instead of storing the secret key of a cryptographic cipher separately from the actual cipher logic, white-box cryptography embeds the key inside the cipher logic. For instance, for AES ciphers, the key can be embedded by multiplication with the T-boxes of each encryption round [12]. However, simply embedding the key in the T-boxes of AES is prone to key extraction attacks since the specification of AES is publicly known. Therefore, WB-AES implementations use complex techniques to prevent key extraction attacks, e.g., wide linear encodings [32], perturbations to the cipher equations [2] and dual-ciphers [21].

The idea behind the white-box approach in [5] is to encode the internal AES cipher logic (functions) inside lookup tables (LUTs). One extreme and impractical instance of this idea is to encode all plaintext-ciphertext pairs corresponding to an AES cipher with a 128-bit key, as a LUT with 2^{128} entries, where each entry consists of 128-bits. Such a LUT would leak no information about the secret-key but exceed the storage capacity of currently available devices. However, this LUT-based approach also works for transforming internal AES functions (e.g. XOR functions, AddRoundKey, SubBytes and MixColumns [12]) to table lookups, which can be divided such that they have a smaller input and output size. Moreover, LUTs can also be used to encode random invertible bijective functions, which are used to further obfuscate the LUTs representing internal AES functions. This leads to an implementation which is much more compact in terms of storage, in the order of megabytes. However, it is also less resilient to cryptanalysis attacks than the single huge LUT instance mentioned before. Nonetheless, such a white-box cryptographic cipher still requires a higher work-factor (i.e. $\leq 2^{22}$ [24]), relative to attacks on systems which store the encryption key separately from the cipher logic [27].

The last decade has seen many new and improved white-box cipher proposals [2, 25, 32, 21]. Several research efforts have also been focused on key extraction of proposed white-box ciphers via cryptanalysis [1, 31, 26, 10, 9]. However, these attacks assume that the location and structure of the LUTs used by the white-box ciphers is known or can be easily recovered from its binary file. Changeware writers must create software, which executed with no administrator privileges can: (1) automatically extract the secret keys from *a large number* of white-box cipher instances one of which is deployed on any victim's machine; and (2) use the corresponding secret keys to change the values of assets on the corresponding machines. In order to counter such automated attacks against white-box ciphers, we employ software diversity, presented in Section 2.2.

2.2 Software Diversity

The intuition behind software diversity stems from biology where biodiversity implicitly serves as a species survival mechanism against disease and viruses [33]. Similarly, it has been shown that software diversity is able to neutralize attacks tailored for a particular software instance, when applied to diverse software instances [13].

Software diversity comes in different flavors, e.g. N-version programming [4], system configuration diversity [17], automated software transformations [14], etc. In our work we use automated software transformations, because they offer a good trade-off between the effectiveness against auto-

mated secret key extraction attacks against white-box cryptographic ciphers and the cost of generating diverse cipher instances (e.g. N-version programming of white-box cryptographic ciphers would have much higher costs).

The seminal work of Forrest *et al.* [13] shows how software diversity can protect against some stack-based attacks (e.g. code-injection). Modern operating systems implement a variant of the ideas introduced in [13], called *address space layout randomization* (ASLR), which diversifies the base addresses of large program objects (e.g. code segment, data segment). However, ASLR cannot counter changeware, which directly changes user editable software assets and uses legitimate OS APIs to inject code into the target software process. Section 3.3 presents the software transformations we employed to generate diverse WB-AES cipher instances, in order to withstand changeware attacks.

2.3 Run-time Integrity Checking

Software self-checking augments code so that it can protect itself against unauthorized modifications. This protection is useful for software which includes functionality that an illegal user may want to circumvent, e.g. license checking. Self-checking is performed by *guards* [3] or *testers* [18]. As the program executes code guards/testers read a range of instructions from memory and compare their hash against a precomputed value. Such techniques are effective against attacks which hot-patch the target software, however they are not effective against changeware which injects code into a process and starts a remote thread.

Jacobson *et al.* [19] define *conformant program execution* as a set of run-time checks on program states, where a state represents elements of a machine that are affected by program execution (e.g. registers and memory). The two elements which characterize the program state are the program counter (PC) and the call stack (currently active stack frames). They dynamically construct a *call multi-graph* (CMG) for a target binary, where nodes represent procedures and arcs represent procedure calls. The CMG is constructed at load time via disassembly, afterwards the program's execution is monitored. At every system-call, their run-time monitor checks for inconsistencies: (1) between the call stack and the CMG and (2) between the PC and the valid program instructions, which are recovered during disassembly. If the procedure call sequence on the stack is not a path in the CMG or if the PC does not point to a valid instruction, then the program execution is non-conformant and it is terminated to prevent code-reuse attacks such as return- and jump-oriented programming.

In contrast, our technique does not construct a CMG for the target binary at load-time. Instead, a subgraph of the CMG is constructed by the vendor before installation on the end-user system. The CMG subgraph can hence be constructed directly from source code, which eliminates the problem of incomplete or incorrect x86 disassembly [11, Chapter 1]. This CMG subgraph only involves the paths which end with procedures that modify software assets. During execution of the target program our checks do not require code disassembly, because they operate directly on the binary. Hence, we do not check if each PC points to a valid program instruction, but instead we compute a single hash of all code segments where each PC points to, and compare this to a value pre-computed by the vendor. Our run-time integrity check arguably has a lower impact on program performance because it is only executed when changes are made to software assets.

3. APPROACH

The scenario in the introduction assumes X to be used by huge numbers of end-users and targeted by changeware developers. In the following we assume that X has in the order of 100 MBs or more of read-only binaries, and that X needs to modify its editable assets at most once per second. We further assume that the code for the modification of the editable assets consists of a fixed set of non-recursive call sequences, i.e. there are no unpredictable call sequences that modify assets. This assumption will be crucial for our authentication approach presented in Section 3.4.2.

Our goal is to protect the integrity of a set of software assets associated with an application targeted by changeware. One straightforward solution is to use a (secret) key to compute a message authentication code (MAC) of the software asset values on the end-user system, whenever they are changed by the target application. This requires embedding the key inside the application binary which is shipped to the end-user. Because cryptographic keys are small chunks of high-entropy data (e.g., an AES key has 128 bits) and application code has lower entropy, the keys can be extracted automatically in linear time with respect to the size of the binary file of the target application [27].

3.1 Protect Key with White-Box AES

To counter entropy-based key extraction attacks, we use the WB-AES technique of Chow et al. [5], which embeds the key into a network of LUTs (Section 2.1). This key is used to compute message authentication codes (MACs) for integrity checking, and to compute encrypted assets for restoration purposes (Section 3.4).

MAC verification requires that the secret key of the WB-AES cipher be persisted between shut-down and start-up of X. This is the case when using WB-AES with an embedded key. Unfortunately, the WB-AES cipher which is stored in a binary located on the end-user system is prone to automated key-extraction attacks [1, 24]. However, these attacks assume that the location and structure of the LUTs of the WB-AES cipher inside the target binary are known. This is a fair assumption given that WB-AES LUTs are large in size and have a high entropy. Nevertheless, a LUT based WB-AES instance consists of thousands of LUTs with different structures and high entropy, which are each used at a specific point (round) of the cipher operation. Therefore, a practical attack would also require identifying which LUTs from the target binary have which function in the white-box cipher [29]. To increase the work-factor for WB-AES LUT identification, in Section 3.2 we present how several *syntactically different* and *semantically equivalent* binary instances of X can be automatically generated using software diversity through randomized obfuscation transformations.

3.2 Prevent Automated Attacks with Diversity

Software diversity can be used to generate a different binary instance for each end-user or generate different instances for groups of end-users (see below). Since the changeware developer does not know which end-user has which binary, s/he must develop an attack effective against all diverse instances of X. We argue that such a universally effective changeware must take a *trial-and-error* approach to-

wards extracting the WB-AES key from a binary, due to the randomized obfuscation transformations applied to every diverse binary instance. Therefore, even if the changeware developer is fully aware of the obfuscation transformation types which are applied, due to randomness of transformation parameters s/he has to search through several possible WB-AES LUT configurations and run a cryptanalysis attack [1, 24] on each configuration. The correctness of every key extracted by a cryptanalysis attack must be checked by using it to encrypt the legitimate settings with a plain AES cipher and verifying that the the ciphertext matches the one computed by the WB-AES cipher.

However, software diversity raises the following concerns:

- *Storage and distribution costs* increase proportionally with the number of diverse release builds of X.
- *Differential updates* are harder to generate and push to end-users, because different instances require different update patches. Moreover, these updates become larger in size, because diverse instances contain many differences relative to each other.
- *Crash analysis servers* run by the software vendor must store the debugging symbols of every existing version of X. Upon receiving a *crash report* containing a snapshot of X's memory, the crash analysis servers must map this report to the corresponding debugging symbols, to perform a correct analysis.
- *Time-stamping binary signatures* is necessary to preserve the validity of the signature for X, even after the signing certificate expires [16]. However, time-stamping requires use of a limited rate Internet service offered by the OS vendor, which is the main bottleneck in case thousands of instances of X require time-stamping their digital signatures.

To address these concerns, we propose to separate the WB-AES cipher code from the source code of X. Therefore, X runs in a separate process from the WB-AES cipher, which will be referred to as *WBCrypto* throughout the remainder of this paper. It can be signed by the vendor of X rather than the vendor of the OS, and it acts as a gateway to any legitimate persistent modifications of assets associated with X, i.e., all asset change requests will be delegated to WBCrypto. We call *WBCrypto proxy* the functions of X which delegate the asset change requests to WBCrypto, i.e., the functions which call the WBCrypto API directly and pass it a string command indicating what changes should be made to which assets.

Yet, even though we now have overcome the above concerns, remember that we assumed X to have huge numbers of users. Generating one WBCrypto instance per user then is prohibitive because of time, storage and electric energy requirements. And even if generating different binaries for such a huge user-base was feasible, it would raise privacy concerns because each end-user could now be uniquely identified by the vendor of X. An improvement of this approach is to generate a smaller number (m) of diverse instances (e.g. several thousands) each being indexed from 0 to $m-1$. A WBCrypto instance with index $i \in \{0, ..., m-1\}$ is distributed to an end-user if the unique system identifier (id) of his machine (e.g. file-system UUID [23], Windows SID [15]) satisfies the following relation: $id \bmod m = i$. This leads to m groups of thousands of end-users having the same binary, which offers anonymity of an end-user in the group of users which have the same key [28] and still raises the bar for changeware.

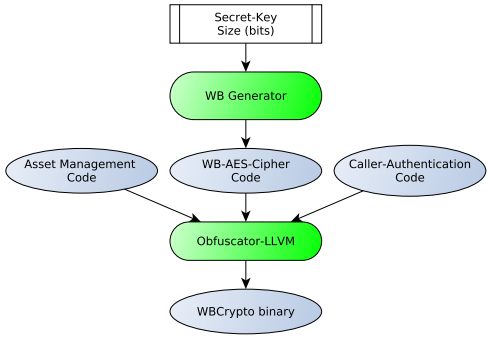

Figure 1: Server-side work-flow

The life-cycle of WBCrypto can be described in two stages. The first stage occurs before WBCrypto is distributed to end-users. It consists of generating diverse WBCrypto instances and occurs on trusted build-servers owned by the software vendor. The second stage occurs after WBCrypto is distributed to end-users. It consists of the operation of WBCrypto on a local system of an end-user.

3.3 Server-Side Generation of WBCrypto

We have implemented a C++ code generator which we call the *WB Generator*. It is hosted on trusted-build servers of the software vendor and can produce white-box AES cipher instances using the LUT-based technique introduced by Chow *et al.* [5]. The WB generator employs both *key-diversity* and *software-diversity*. Key-diversity implies generating different random keys, to mitigate the impact of successful key extraction from one application instance. Specifically, if an attacker extracts the key embedded in his/her WB-AES instance, s/he will not be able to use the same key to manipulate software assets of all other end-users of X (or, if keys are created for groups of users, to manipulate assets in different groups). Software-diversity is employed via data and control-flow obfuscation transformations which increase the attacker's effort for extracting the position and structure of the WB-AES LUTs from any binary instance. Specifically, if an attacker is able to reverse engineer his own copy of WBCrypto and extracts the offset values of the WB-AES LUTs, s/he will not be able to use the same offsets to extract the LUTs from (all) other instances of WBCrypto.

Our current WB generator implementation is limited to the AES cipher with various key sizes (128, 192, 256 bits). In future work we plan to incorporate different block ciphers (e.g. DES, Blowfish) and other WBC techniques [2, 25, 32, 21]. The WB Generator is OS independent and can be used to generate diverse WB-AES ciphers.

Figure 1 shows the work-flow for generating a WBCrypto module on the server-side. The upper part shows the WB generator, which takes as input the key size of the AES cipher. The WB generator first generates a random key and subsequently the LUTs of the WB-AES cipher for the corresponding key. Finally, the WB generator writes the LUTs and instructions for WB-AES en-/de-cryption to a C++ source file. Remember that there is one key, and hence one WBCrypto instance, per group of users.

In order to prevent automatic retrieval of LUTs and thus automatic extraction of the secret key, a second step is to obfuscate the source code using the transformations presented

in the following paragraphs. The output of the WB generator is a C++ source file which must be compiled together with *asset management* code and *caller authentication* code (see Section 3.4). The outputs of the server-side work-flow from Figure 1 are diverse WBCrypto binaries which are shipped to random groups of end-users.

Randomization of Unused Nibbles.

WBCrypto instances contain over a thousand LUTs which represent randomly encoded versions of the XOR function, which have an 8-bit input and a 4-bit output. For efficiency reasons, the possible output values are stored in 1 byte chunks instead of packing two 4-bit values inside of one byte. This means that such lookup tables may be easily identified in the data segment of the WBCrypto binary because they contain values of the form `0x0?`, i.e. only the lowest nibble is used. An attacker could use this pattern to automatically identify the position of the XOR tables in a WBCrypto binary, with high probability of success. Therefore, we obfuscate the representation of these LUTs by transforming their high-nibble from zeros to random values.

Interleave LUTs with Random Data.

In the C++ output of the WB generator, a WB-AES LUT is represented by a statically initialized multi-dimensional array. This means that the bytes corresponding to LUTs are placed in the data segment of the compiled binary. Therefore, they are not affected by ASLR. Hence the LUTs will always be loaded at the same virtual address, relative to the base address of WBCrypto's process memory. An attacker who finds these offsets can use them to extract the LUTs from any WBCrypto binary instance.

We employ an obfuscation technique that adds randomly sized statically initialized arrays containing garbage data in between the initialization of the LUTs used by the WB-AES cipher. Due to the high entropy of the LUTs used by the WB-AES cipher, the added garbage arrays cannot be distinguished from the WB-AES LUTs using entropy analysis. This obfuscation technique enlarges the search space of the attacker directly proportional to the number and size of the garbage arrays. This leads to a trade-off between the size of the WBCrypto binary and the size of the search space of the attacker, which we discuss in Section 4.

Control-Flow Obfuscation.

For the purpose of control-flow obfuscation we employ *instruction substitution* [7], *opaque predicate insertion* [8] and *control-flow flattening* [22] techniques. These techniques are implemented by the open-source compile-time obfuscation engine *Obfuscator-LLVM* [20] (lower green-rounded-rectangle in Figure 1). These obfuscation transformations change the form of references to the WB-AES LUTs inside the code segment of WBCrypto. We also employ anti-disassembly techniques [11, Chapter 21], to prevent changeware from disassembling the WBCrypto binary and extracting the address of the WB-AES LUTs from assembly code.

3.4 Client-Side Operation of WBCrypto

WBCrypto computes and stores a MAC and possibly the ciphertext of software assets upon installation and whenever these values are *written* by X. On start-up, or more generally, whenever the asset is *read*, WBCrypto computes a new MAC for comparison purposes. For restoration purposes, it may also compute the ciphertext of the current values of X's software assets. If the MACs differ, then changes were made to its assets while X was offline or, more generally, in-between the last read and write operations on assets, and the end-user is notified. If the end-user does not agree with the changes, then s/he can restore the previous version of the assets by decrypting the last good known version. If changeware repeatedly changes the assets, each modification will lead to a notification, which is likely to quickly annoy the user. Therefore, the replacement with the known good value can also be done automatically. If there is only a MAC but no known good version, then default settings can be restored. It is also possible to retrieve any last good known version of the assets if a backup copy of that version of the encrypted software assets has been stored on cloud storage by the end-user. Because they are encrypted, asset values are not disclosed to the cloud service provider. Such a backup also ensures that software assets can be restored even if changeware deletes the ciphertext from local storage.

Remember that having WBCrypto as a self standing software module has several advantages compared to integrating the WB-AES code inside X's binary. WBCrypto is relatively small in size (a few MBs) compared to X and can therefore be built, shipped and updated separately. The crash analysis process is unaffected since only WBCrypto is diversified, not X itself. Since WBCrypto is only used by X, its authenticity can be ensured by a digital certificate signed by the vendor of X and not by the OS vendor. The WBCrypto code verification key—which is not the secret-key embedded in WBCrypto—can therefore be hard-coded inside X's binary, which eliminates the need for time-stamping the binary signature of all the diverse instances of WBCrypto by the service of the OS vendor which is a potential bottleneck. We may thus assume that X is able to verify the integrity of the WBCrypto binary.

3.4.1 Malicious Calls to WBCrypto

However, separating the WB-AES cipher and X into distinct binaries also has disadvantages. Most importantly, it opens up another attack vector. The interface of WBCrypto, which previously was only internally accessible to X, is now accessible by changeware. To prevent calls performed by changeware to the WBCrypto interface, we need a run-time checking mechanism which can discriminate between calls intended by the software vendor (*benign calls*) and other calls which we consider to be *malicious calls*.

One approach towards authenticating the caller of the WBCrypto interface is to simply check if the calling process has several characteristics of X, e.g. the loaded library modules are those which are supposed to be loaded by X. However, not all calls originating from a particular process are necessarily benign. Consider a simple defense mechanism which merely checks whether or not the calling process is in a set of benign processes. On Microsoft Windows OSs, such a defense mechanism is vulnerable to the attack illustrated in Figure 2, where a malicious process (M) injects code into the benign process (e.g. X) and then starts a remote thread which surreptitiously calls WBCrypto proxy functions. In this case the call will appear to come from a benign process, however it should not be executed by WBCrypto, because it is not the behavior intended by the vendor of X.

Code injection and remote thread creation do not require exploiting a vulnerability in X, because the Windows API

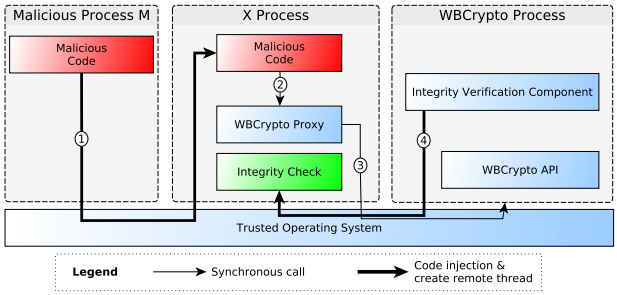

Figure 2: Malicious Call-Path and Caller Authentication

offers functions which can perform such actions on any process running under the same OS user, i.e. `WriteProcessMemory` and `CreateRemoteThread`.[1]

3.4.2 Caller Authentication

A skillful attacker could develop changeware such that it has the same call-stack structure as X, i.e. the return addresses on the call-stack of changeware point to the same code offsets as the return addresses on the call-stack of X. On the other hand, an attacker could also develop changeware such that its code pages would contain the code pages of X, yet its call-stack structure would be different. However, if both the code and the call-stack structure of changeware are the same as in X, then the changeware can be considered equivalent to X, hence benign software. Moreover, we do not exclude changeware which tampers with the code of X after it is loaded in process memory if that code page is writable. Therefore, we compare precomputed *known good values* with both (1) the call-stack structure and (2) the code to where the return addresses on the call-stack point to. This allows us to effectively detect malicious calls to the WBCrypto API as described in Section 3.4.1.

We define a *call-path* as a chain of (synchronous, blocking) function calls (possibly across thread or process boundaries) such that the last function in this chain is a WBCrypto proxy function. A call-path is uniquely identified by a fixed size hash value computed from the concatenation of all the relative return addresses on the call-stacks of the execution threads which performed the function calls, and the memory pages to which the return addresses on the stacks point to.

The set of all *benign call-paths*, representing the intended behavior of X, is fixed by its vendor before deployment on the end-user systems. This set is distributed as a read-only white-list signed by the software vendor, created by:

1. Generate the call-graph of X, e.g., by using the Callgrind tool of Valgrind (`http://valgrind.org/`);
2. Select the paths in the call-graph which end in a call to the WBCrypto API, i.e. WBCrypto proxy functions;
3. Compute and store hash values of invariants of all selected paths (see below).

Any call-path hash value not included in this white-list is considered to be a *malicious call-path*.

At runtime, the *caller authentication code* of WBCrypto (mid-right blue-oval in Figure 1), computes the call-path hash for every API call it gets from a process P, to the function which sets the value of a software asset. If the call-path hash value is in the white-list, then WBCrypto executes the API call it received from P, otherwise it discards it. This guarantees that all of the attacks presented in Section 3.4.1 will be unsuccessful in surreptitiously changing the values of the software assets.

The caller authentication implementation consists of 3 phases, executed whenever any process (denoted P) makes a call to the WBCrypto API, sending it a string command v, which indicates persistent changes to software assets should be made. Fighting fire with fire, in the first phase WBCrypto injects an *integrity check* routine into the calling process X, using the API of the trusted OS, i.e. step 4 in Figure 2. In the second phase, the previously injected code is executed in a dedicated execution thread ($T1$) inside P. This thread performs the following steps:

1. To compute the hash value of the current call path, which we will later compare to the precomputed legal paths, we first need to identify the thread ($T2$) inside the process memory of P which called any known WBCrypto proxy function, from software X's binary. This is done by traversing the stack of each thread in P, searching for a return address pointing inside any WBCrypto proxy function (denoted S), having a pointer to the string command v on the same stack frame as the return address to S. This guarantees that $T2$ ends with a synchronous call to the WBCrypto proxy function with argument v (i.e. $S(v)$), corresponding to the command received by WBCrypto, which triggered caller authentication. If $T2$ is found, we continue with step 2, otherwise if $T2$ is not found, caller authentication stops and denies the authentication.
2. If $T2$ is identified, then this is the thread which made the call to the WBCrypto API for persistent asset changes. We extract the *absolute* return addresses from each stack frame by iterating over $T2$'s stack one word at a time, starting from the stack base address. On each stack frame we will find the value of the previous frame pointer (EBPs), which we recognize because it points to lower memory addresses on the same stack. The value of the absolute return addresses is the first double-word under the EBP.
3. The *relative* return address is obtained by subtracting the base address of the dynamic library module where the relative address points to, from the absolute return address. The relative return address is relevant because it is invariant with regard to different executions of the target software on the same OS version.
4. We compute a hash: (a) of the code pages where each absolute return address points to and (b) of the relative return address values. This hash uniquely identifies

[1]This kind of attack is also possible on current Debian Linux and was also possible on Ubuntu Linux before August 2011, via the `ptrace` system call, which is generally used by debuggers. Current Ubuntu Linux versions use a system flag called `ptrace_scope`, which by default allows a non-root process to attach via `ptrace` only to its child processes or children of the debugger, see `https://wiki.ubuntu.com/Security/Features#ptrace`. Nonetheless, this default setting is often changed by a root user to enable operation of some applications, e.g. Mono applications, Qt Creator, GNU Debugger, etc. Moreover, we do not exclude buffer overflow exploits for X, which aim at changing the control flow of X such that it calls the WBCrypto API in order to surreptitiously change the values of software assets.

the stack structure with relative return addresses and the code associated with $T2$'s stack.

5. Because call-paths may cross thread boundaries, we recurse on step 1. This time, instead of searching for a WBCrypto proxy function, we search for functions which may have triggered the WBCrypto API call of $T2$. These functions are known by the vendor of X because they constitute the intended behavior of X. If we did not consider call-paths which span over different threads, an attacker could take advantage of this and trigger a call to the WBCrypto API via a legitimate thread $T2$, which would be seen as a benign call by WBCrypto.

The output of this algorithm is a hash of the concatenated hash values from all threads in one call-path.

In the third phase, WBCrypto waits for thread $T1$ to finish and retrieves its return value, i.e. the call-path hash value. It is subsequently compared to white-list entries.

3.4.3 Message Authentication

We have implemented *caller authentication* described in Section 3.4.2 for the Chromium open-source web-browser, where changeware attacks are represented by hijacking browser user preferences. Our implementation is specific to Microsoft Windows OSs, for which browser hijacking changeware often comes pre-packed with other software such as toolbars and are therefore installed by mistake by end-users.

Our implementation detects if a malicious thread directly/synchronously calls the WBCrypto proxy functions in Chromium. However, if changeware is able to asynchronously post a call to WBCrypto via a legitimate call-path, then the caller authentication will consider this call benign and it will execute it. This problem occurs for asynchronous (non-blocking) function calls, which break a call-path in separate synchronous parts. This causes every call before and including the last asynchronous function call to be discarded from the call-path, which is verified via caller authentication. We call this problem *message authentication*, because the asynchronous function calls may be seen as messages which are sent/posted from callers to callees.

We envision an attack where a malicious thread can surreptitiously change asset values by delegating this task to a legitimate thread, via asynchronous message passing. In the first step of this attack, changeware injects code into the target process. Subsequently this injected code posts a message to a legitimate thread, hence delegating a call to the WBCrypto proxy functions.

We acknowledge that asynchronous inter-thread communication is frequently encountered in applications, because such an architecture offers better performance and UI responsiveness. Therefore, we also implemented a proof-of-concept mechanism for message authentication.

The motivation for message authentication is the need to verify that only legitimate threads possibly asynchronously posted the message which led to a WBCrypto API call. For this purpose, legitimate threads are identified via application internal IDs, instead of via OS assigned thread IDs. To query if a legitimate thread sent a message which led to a WBCrypto API call, we add a private data structure (Θ_x) to the thread local storage of each browser thread (T_x). The memory address of thread local storage is assumed to be unknown by other threads, because it is randomly chosen by the OS kernel. A malicious thread hence cannot locate and modify Θ_x belonging to a legitimate thread T_x. Θ_x stores *only* those messages (denoted θ) posted by this thread that contain API calls to WBCrypto.

To notify WBCrypto about which Chromium thread made the call, we modify the WBCrypto proxy functions to send the internal ID indicated by the thread (T_x) which posted the message. Note that this ID value can be spoofed by a malicious thread, which does not have this ID such that a message appears to have been posted by a benign thread.

Due to message asynchronicity each hash value in the call-path white-list needs to be associated with a set of benign threads, which are allowed to perform the corresponding call-path. In our implementation we only allow the Chromium *main thread* to post messages containing API calls to WBCrypto. We ensure this by only including hashes corresponding to legitimate call-paths in the white-list.

The last step enables WBCrypto to compute the call-path hash value only if T_x is associated with any call-path in the white-list. Using the previous modifications, the integrity checking module in our caller-authentication thread inside the Chromium process can post a message to T_x "asking" if it sent θ. A malicious thread will never be "asked" anything, because Chromium uses internal IDs (not OS assigned thread IDs) to identify special browser threads such as the main thread. Using a whitelist of internal thread IDs, we can ensure that only threads which have an internal ID are allowed to post messages which call WBCrypto proxy functions. T_x receives the "question" from the integrity checking module and replies affirmative only if θ is in its local Θ_x. If T_x replies affirmative, then the integrity check thread continues to compute the call-path hash value as in the original caller authentication algorithm, otherwise is returns an invalid hash value to WBCrypto.

4. EVALUATION

4.1 Performance Evaluation

4.1.1 WB Generator

We tested the WB generator on an average system with 2 CPU cores and 8 GB of RAM. We generated thousands of WB-AES instances having distinct key sizes and different numbers of random tables of various sizes added to them. The maximum and median sizes of the C++ files and compiled WB-AES instances are shown in Table 1 as a function of the cipher key size. The sizes are larger than the tens of kilo-bytes needed by a standard AES implementation. Compressing them would only negligibly reduce their size, because of the LUTs' high entropy. However, compared to the total size of Chromium binaries, it is roughly 2.5%. The size of the WB-AES instances in fact motivated our respective general space assumption in Section 1.

The maximum and median times to generate and compile the C++ WB-AES instances are shown in Table 2 as a function of the key size. We believe it is possible to significantly improve the median generation and compilation time (ca. 1 min.) by a multi-threaded implementation of the WB generator, which we leave to future work.

We also measured the run-time performance of all the generated WB-AES instances. The time needed to WB encrypt a file grows linearly with the size of the plaintext. The average time needed to WB encrypt 1 mega-byte of data using

	Max C++	Max bin	Med C++	Med bin
AES-128	115	20	15	2.6
AES-192	164	27	18	4.5
AES-256	166	26	20	4.8

Table 1: Size (**mega-bytes**) of C++ and binary WB-AES instances (Chromium binaries ≈100MBs)

	Max Gen	Max Cpl	Med Gen	Med Cpl
AES-128	475	154	40	12
AES-192	1075	326	37	16
AES-256	1049	239	32	22

Table 2: Time (**seconds**) to generate C++ WB-AES instances

AES-128, AES-192 and AES-256 is 3.25 seconds, 3.56 seconds, respectively 4.11 seconds. This average is significantly slower than using the latest version of *openssl* to encrypt the same file, which takes 25, 27, respectively 30 milliseconds. This does not pose any problems if we use our approach to check integrity breaches (compute and compare MACs), but not necessarily repair them (compute and compare encrypted files), see Section 3.4. Moreover, this does not pose serious problems under the assumption of Section 1 that asset files are rarely written. Both clearly is the case for our Chromium case-study where (1) we encrypt only a few kilo-bytes, or even far less if we use our approach solely to compute a MAC; and (2) settings files are changed rarely.

4.1.2 WBCrypto

Our PoC implementation only influences executions which trigger a call to the WBCrypto API. Other executions are not impacted, which keeps resource consumption minimal.

We have measured the time needed to perform both caller authentication and message authentication on a machine having an Intel Xeon E5645 CPU with 6 cores running at a frequency of 2.4 GHz. The Microsoft Windows 7 OS load has been set to an average value of 100 running processes containing over 1000 threads in total. The results showed that caller authentication execution time averages at 134 milliseconds after a few hundred runs, with a maximum time of 145 milliseconds. Computing the hash over the code pages to where the return address points to is linear with respect to the code size and takes about 48% of the total execution time. Iterating over the portable executable (PE) relocation table and setting the indicated offsets to a constant fixed value takes about 42% of the total execution time. This step is necessary to ensure that hashes computed over the code pages do not contain random addresses set by ASLR. The remaining 10% of the execution time is required for finding the thread which made the call to the WBCrypto API, and iterating over its call-stack. The execution time difference between message authentication and caller authentication varies between 19 and 50 milliseconds.

Since user preference changes are an infrequent event in every-day browser usage, even such a delay arguably does not have negative impact on the end-user experience, because it does not freeze the browser UI and it is small enough to pass until the user gets a chance to notice it.

4.2 Security Evaluation

The attack tree in Figure 3 shows the attack space for a changeware developer. The root of the tree represents the goal to persistently change the values of software assets without being detected. Effective changeware may: (1) use the secret key to re-compute asset MAC values after modifying

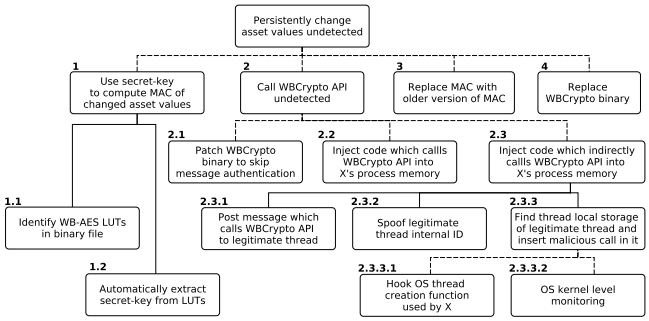

Figure 3: Attack Tree (dashed OR-edges, solid AND-edges)

them, (2) call the WBCrypto API such that it is not detected by the integrity check of message authentication, (3) replace the asset value and its MAC with an older value which fits the attacker's interest or (4) replace the WBCrypto binary with a rogue or cracked version. The following sub-sections present each of these attacks in detail.

4.2.1 LUT Search-Space Increase

Assume that a browser-hijacking changeware writer is able to reverse-engineer the WBCrypto instance on his local system such that s/he knows: (1) WBCrypto uses the WB-AES cipher of Chow *et al.* [5] (2) the size and structure of the useful LUTs and (3) the additional obfuscation transformations described in Section 3.3. Even with this knowledge the changeware developer must, for each instance of WBCrypto, implement a search algorithm to identify the memory offset of the LUTs in the data segment of the WBCrypto binary (node 1.1 in Figure 3). Let n be the total size of the added garbage LUTs in bytes and let k be the number of useful LUTs of the WB-AES cipher. On average, changeware has to perform a key-extraction attack $(n+k)!/2$ times, because the garbage LUTs are randomly interleaved with the useful LUTs. Therefore, there is a trade-off between security and size: increasing n will also increase the size of WBCrypto. In our PoC implementation we group the 3008 WB-AES LUTs together according to the 4 types of LUTs presented in [5]. Half of these LUTs are used for encryption, while the other half are used for decryption. Therefore, the C++ output by our WB Generator always contains $k = 8$ statically initialized arrays which group together all the WB-AES LUTs, plus a random number of randomly sized garbage arrays.

Setting the total size of the garbage arrays to $n = 100$ bytes, an attacker has to execute the WB-AES cryptanalysis attack of [24] for $(100 + 8)!/2$ different configurations of the WB-AES LUTs. Even if a changeware search algorithm would be improved such that it could locate the WB-AES LUTs in logarithmic time, this would still require over 166 days on a 2,4 GHz CPU. For our experiment described in Section 4.1.1 we have generated WB-AES instances with the values of n up to 100 MBs, which would require hundreds of thousands of years of execution time on the above CPU.

4.2.2 Malicious Calls to WBCrypto

If a changeware developer believes that the protection offered by the obfuscation and diversity techniques are difficult to attack, s/he may attempt to develop changeware that calls the WBCrypto API without this being detected as coming from a malicious origin.

Changeware could be built such that it patches the WBCrypto binary to skip the message authentication integrity check (node 2.1). This attack is detected by the signature verification performed by X before it launches WBCrypto as its child process. Remember that in Section 3.4 the WBCrypto binary is signed by the vendor of X and the signature verification key is embedded in X's binary.

Another attack would be to inject code inside of X's process memory, which then calls the WBCrypto proxy functions as described in Section 3.4 (node 2.2). This attack is detected by our caller authentication algorithm (see Section 3.4.2), because the hash of the malicious call-path will not be in the white-list of WBCrypto. Moreover, any (a) synchronous call originating directly from a malicious process will also be detected, because the return value of the integrity check will not be in the set of benign call-paths. We have implemented the direct (synchronous) and indirect (asynchronous) calls to WBCrypto as demonstrator changeware examples using the Windows native API. These samples are able to surreptitiously change Chromium user preferences if caller authentication is not performed. They are all detected by caller authentication, with the exception of changeware which asynchronously posts a message to a legitimate thread of X (node 2.3). However, such changeware is detected by message authentication (see Section 3.4.3) even if it spoofs its internal thread ID (node 2.3.2).

Message authentication relies on the fact that attribute values stored in the thread local storage of one thread are not accessible by other threads. Therefore a malicious thread cannot directly add a task to the locally stored task list of a legitimate Chromium thread (node 2.3.3).

To overcome this mechanism, the changeware would need to monitor the inter-thread communication inside the Chromium browser process. One possible way of monitoring would be at the OS kernel level (node 2.3.3.1). However, that would require changeware to have root privileges, which was excluded in Section 1. If changeware had root privileges it could directly modify the Chromium binaries and disable the OS code signature verification "alarms." For instance, changeware with root privileges could disable *User Account Control* in the Microsoft Windows OS, which notifies the user if a binary signature is valid.

Another possible attack is to hook the thread creation system calls (e.g. `BaseThreadInitThunk` from kernel32.dll) and insert functionality which would leak the desired information (i.e. Θ_x's address) from the legitimate thread of X (node 2.3.3.2). However, the integrity check performed via *caller authentication* would detect this hook, via the call-stack inspection of the legitimate thread, because this change would return an invalid hash value to the WBCrypto module.

4.2.3 Replay Attacks

Usually, some asset values change more frequently than others. We partition an asset file into a set $\{p_0, p_1, \ldots, p_n\}$, $n \geq 0$ and compute a ciphertext for each part $c_i = \epsilon_k(p_i)$. This partition can be as fine grained as needed by the application. *Replay attacks* (node 3) in this context consist of changeware installed on an end-user system: (1) recording several asset values and their corresponding MAC values and (2) eventually replacing a current asset value and its MAC with a previously recorded value.

For instance if p_i is set by the target software as a legitimate asset value, then changeware interested in keeping this value records its MAC (c_i). Later if the end-user decides to switch the asset value to p_i', changeware can surreptitiously change this asset value back to p_i. To prevent such replay attacks, we use the WB-AES cipher to compute one single MAC value of the concatenation of: (1) all ciphertexts and (2) the time-stamp when the asset file was last modified. The MAC is computed as $h = H(\epsilon_k(c_0 \mid c_1 \mid \ldots \mid c_n \mid time))$, where H is a hash function such as SHA-256. Since changeware does not have root privileges it cannot tamper with the modification time of a file, which is managed by the OS.

4.2.4 Replace WBCrypto Binary

Replacement of the WBCrypto binary with a rogue binary, which mimics the behavior of WBCrypto, is prevented by the code signature of the vendor of X (see Section 3.4). Nonetheless, an attacker could semi-automatically cryptanalyze the WBCrypto binary instance deployed on his own machine and extract the secret key. Afterwards, he can build changeware which replaces the WBCrypto of any end-user of X, with his cracked version for which the secret key was extracted. This attack is prevented due to WBCrypto binary binding to end-user devices via the unique identifier (id) provided by the OS. As described in Section 3.2, the WBCrypto module instances are indexed from 0 to m and an instance is deployed only if its index i is congruent to the end-users' id modulo m. On startup, X first performs a system call to retrieve the id and then checks whether the signature of WBCrypto is valid for the hash of the current WBCrypto binary concatenated with $id \bmod m$. Therefore, the attacker can replace WBCrypto instances only on end-user devices that have the same id. However, the WBCrypto instances on devices with the same id are identical, hence there is no benefit in replacing them. Moreover, an attacker does not know which end-users have the same id as him.

5. DISCUSSION AND LIMITATIONS

5.1 Applicability of Results

Our solution does not aim to defend against changeware which performs changes in the process memory of the target software, i.e., against changeware that makes dynamic but not persistent changes to software assets. However, we believe that caller authentication (see Section 3.4.2) and message authentication (see Section 3.4.3) can be integrated into the target software in order to protect against such non-persistent changes to process memory.

One example of non-persistent changes to process memory, which can be detected by caller authentication, is an *import address table* (IAT) patching attack which is frequently employed by malware to portable executables on MS Windows. IAT entries can be seen as pairs of function name and pointer to function code. IAT patching replaces the legitimate pointers to function code by pointers to injected (malicious) code. It is detected by message authentication because the hash of the call-path involving any patched function would not match the hash from the white-list.

One limitation of the caller authentication mechanism is its dependency on the code of the libraries needed for benign call-path hash computation. The benign call-path hash values included in a white-list are specific to the version of the libraries where functions on the call-path are defined. Since a part of these libraries are provided by the OS vendor and others by the vendor of the target software (X), the

white-list must be updated for each update of the OS or of X, i.e. around twice per month. This means that there is not one unique white-list for all end-users. Instead there is a white-list specific to each compatible version of libraries of the OS and the libraries of X. Whenever X starts up, it checks if the library file versions associated with the white-list have changed due to updates. If so it will request an update of the white-list to the current library versions. The white-list is a read-only asset and is signed by the vendor of X and distributed/updated separately from WBCrypto. Therefore, WBCrypto is not updated every time the white-list is updated. WBCrypto simply checks if the white-list was tampered with by verifying its signature using the public verification key of vendor X, and then uses the contents of the white-list during caller and message authentication.

5.2 Alternative Solutions

An alternative to our solution is implemented by modifying the OS kernel. We avoided this approach because we want our solution to function without changes to the underlying infrastructure on which the target software runs. However, if changing the OS kernel is possible, we argue that WBC and software diversity are no longer needed, because the kernel can protect secret-keys under root privileges such that they are not accessible to changeware.

In order to ensure that each application installed on the OS has a different key, the kernel would generate a unique random secret-key, which is persistently stored under root/ administrator privileges, i.e. not accessible to changeware and is permanently associated only with this application. An application does not directly get access to its unique key, but instead the OS offers a cryptographic API (similar to the one offered by WBCrypto), which an application has to provide the data for which to compute a MAC value. The OS verifies which binary was used to launch the process calling this API and uses the secret-key associated with it to compute the MAC. To prevent changeware attacks by code injection and remote thread creation (as described in Section 3.4), the OS kernel performs caller authentication of any thread which calls this API. The white-list of benign call-paths is safely hard-coded in the target software binary signed by the OS vendor before distribution to end-users.

Another alternative solution is to augment the OS access-control mechanism such that upon installation certain software assets can be marked as editable only by a set of applications, i.e. not by any application running under the same OS user. This kind of application-centric access control would also be able to prevent changeware from writing to another process' memory, i.e., code injection.

These solutions would eliminate the generation, storage and distribution costs for the target software vendor. Moreover, the run-time performance would be improved by using a classical cryptographic cipher implementation instead of a diversified white-box version of the same cipher.

6. CONCLUSIONS AND FUTURE WORK

We have presented a novel software-based solution against changeware, i.e. malware with no root/administrator privileges, which performs surreptitious persistent changes to software assets which are not protected by the OS code signature verification mechanism. Our solution combines techniques from the fields of software obfuscation, software diversity and run-time checking. The solution can be applied by the vendor of software targeted by changeware; its function is transparent to end-users. This solution is effective under the assumption that it is unfeasible for an attacker to manually construct attacks for a large set of diverse instances of the target software, i.e. for a large part of its user base. Additionally we require the code for the modification of the editable assets to be a fixed set of non-recursive routine call sequences, i.e. there are no unpredictable call sequences that modify assets.

Performance of the target software is slightly affected due to additional time needed to perform caller authentication and white-box encryption. However, this performance overhead is in the order of a fifth of a second, which is acceptable assuming that persistent changes to software assets occur rather infrequently. Moreover, our solution requires non-negligible production and distribution resources for WBCrypto, on the software vendor's side. However, the size of WBCrypto is under 5 MBs and poses a much smaller overhead than re-distribution of the entire target application which we assumed to be in the order of 100s of MBs. We also assumed a huge user base because otherwise changeware developers are unlikely to invest into an attack.

The software diversity and obfuscation transformations offered by the WB Generator, used to generate WBCrypto are application and OS independent. The caller authentication algorithm can be applied to any application if: (1) the OS allows code injection and execution of this code in the target's process memory and (2) the target application permits blocking function calls for the period needed to perform caller authentication. Currently Microsoft Windows OSs allow code injection via system calls. This is not allowed on all Linux distributions. The message authentication mechanism can be implemented for applications which mandate that asset values should be changed only via special threads which can be internally identified, without using the OS assigned thread ID.

Caller- and message-authentication were necessary because the WBCrypto code could not be directly integrated in the code of the target application due to technical and organizational concerns such as: storage and distributions costs, differential updates, crash analysis and most importantly time-stamping binary signatures. If we had taken a less specific approach in our work, such concerns would not have been considered, which means that integrating an obfuscated and diversified WB-AES cipher into the target software would have been a viable solution. Our work shows that such a viable solution is not applicable in practice if it cannot be easily integrated in the existing software engineering processes of the target software. Therefore, we adapted the solution such that it is practicable.

There are several possible future research directions for the current work. The WB generator can be improved by adding additional diversity dimensions, such as *cipher diversity*, i.e. implementing additional block ciphers (e.g. DES, Blowfish, etc.) and more WBC techniques [2, 25, 32, 21]. The performance overhead of caller authentication can be improved by tuning the memory integrity checks or finding cheaper substitutes for the most costly operations. The performance of the WB Generator can be improved such that it offers faster builds and smaller sized WBCrypto instances. Finally, the most interesting problem we are currently working on is quantification of the attacker effort added by software diversity and obfuscation in this context.

7. REFERENCES

[1] O. Billet, H. Gilbert, and C. Ech-Chatbi. Cryptanalysis of a white box AES implementation. In *Selected Areas in Cryptography*, number 3357 in Lecture Notes in Computer Science, pages 227–240. Springer Berlin Heidelberg, Jan. 2005.

[2] J. Bringer, H. Chabanne, and E. Dottax. White box cryptography: Another attempt. *located at, last visited on Jul*, 22(2011):14, 2006.

[3] H. Chang and M. J. Atallah. Protecting software code by guards. In *Security and privacy in digital rights management*, pages 160–175. Springer, 2002.

[4] L. Chen and A. Avizienis. N-version programming: A fault-tolerance approach to reliability of software operation. In *Proc. 8th IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-8)*, pages 3–9, 1978.

[5] S. Chow, P. Eisen, H. Johnson, and P. C. V. Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, number 2595 in Lecture Notes in Computer Science, pages 250–270. Springer Berlin Heidelberg, Jan. 2003.

[6] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[7] F. B. Cohen. Operating system protection through program evolution. *Computers & Security*, 12(6):565–584, Oct. 1993.

[8] C. Collberg, C. Thomborson, and D. Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[9] Y. De Mulder, P. Roelse, and B. Preneel. Cryptanalysis of the Xiao-Lai white-box AES implementation. In *Selected Areas in Cryptography*, pages 34–49, 2013.

[10] Y. De Mulder, B. Wyseur, and B. Preneel. Cryptanalysis of a perturbated white-box AES implementation. In G. Gong and K. C. Gupta, editors, *Progress in Cryptology - INDOCRYPT 2010*, number 6498 in Lecture Notes in Computer Science, pages 292–310. Springer Berlin Heidelberg, Jan. 2010.

[11] C. Eagle. *The IDA pro book: the unofficial guide to the world's most popular disassembler*. No Starch Press, 2011.

[12] P. FIPS. 197: Advanced encryption standard (aes). *National Institute of Standards and Technology*, 2001.

[13] S. Forrest, A. Somayaji, and D. Ackley. Building diverse computer systems. In *Operating Systems, 1997., The Sixth Workshop on Hot Topics in*, pages 67–72, 1997.

[14] M. Franz. E unibus pluram: massive-scale software diversity as a defense mechanism. In *Proceedings of the 2010 workshop on New security paradigms*, NSPW '10, pages 7–16, New York, NY, USA, 2010. ACM.

[15] S. Govindavajhala and A. W. Appel. Windows access control demystified. Technical report, Department of Computer Science, Princeton University, 2006.

[16] S. Haber and W. S. Stornetta. How to time-stamp a digital document. In *Proceedings of the 10th Annual International Cryptology Conference on Advances in Cryptology*, CRYPTO '90, pages 437–455, London, UK, UK, 1991. Springer-Verlag.

[17] M. A. Hiltunen, R. D. Schlichting, C. A. Ugarte, and G. T. Wong. Survivability through customization and adaptability: The cactus approach. In *DARPA Information Survivability Conference and Exposition, 2000. DISCEX'00. Proceedings*, volume 1, pages 294–307. IEEE, 2000.

[18] B. Horne, L. Matheson, C. Sheehan, and R. E. Tarjan. Dynamic self-checking techniques for improved tamper resistance. In *Security and privacy in digital rights management*, pages 141–159. Springer, 2002.

[19] E. R. Jacobson, A. R. Bernat, W. R. Williams, and B. P. Miller. Detecting code reuse attacks with a model of conformant program execution. In *International Symposium on Engineering Secure Software and Systems (ESSoS)*, pages 1–18. Springer Berlin Heidelberg, 2014.

[20] P. Junod, J. Rinaldini, and J. Wehrli. Obfuscator-LLVM. `https://github.com/obfuscator-llvm/obfuscator`, 2014. GitHub repository.

[21] M. Karroumi. Protecting white-box AES with dual ciphers. In K.-H. Rhee and D. Nyang, editors, *Information Security and Cryptology - ICISC 2010*, number 6829 in Lecture Notes in Computer Science, pages 278–291. Springer Berlin Heidelberg, Jan. 2011.

[22] T. László and Á. Kiss. Obfuscating c++ programs via control flow flattening. *Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica*, 30:3–19, 2009.

[23] P. Leach, M. Mealling, and R. Salz. A Universally Unique IDentifier (UUID) URN Namespace. RFC 4122 (Proposed Standard), July 2005.

[24] T. Lepoint, M. Rivain, Y. De Mulder, P. Roelse, and B. Preneel. Two Attacks on a White-Box AES Implementation. In *Selected Areas in Cryptography–SAC 2013*, pages 265–285. Springer, 2014.

[25] W. Michiels and P. Gorissen. Mechanism for software tamper resistance: an application of white-box cryptography. In *Proc. ACM workshop on Digital Rights Management*, pages 82–89, 2007.

[26] W. Michiels, P. Gorissen, and H. D. L. Hollmann. Cryptanalysis of a generic class of white-box implementations. In R. M. Avanzi, L. Keliher, and F. Sica, editors, *Selected Areas in Cryptography*, number 5381 in Lecture Notes in Computer Science, pages 414–428. Springer Berlin Heidelberg, Jan. 2009.

[27] A. Shamir and N. Van Someren. Playing 'hide and seek' with stored keys. In *Financial cryptography*, pages 118–124, 1999.

[28] L. Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(05):557–570, 2002.

[29] SysK. Practical cracking of white-box implementations. In *Phrack Magazine*. Phrack Inc., 2012. Volume 0x0e, Issue 0x44, Phile #0x08 of 0x13.

[30] Trusted Computing Group. Trusted Platform Module (TPM) Specifications. Online at `https://www.trustedcomputinggroup.org/developers/`

`trusted_platform_module/specifications`. Accessed on: 14-07-2014.

[31] B. Wyseur, W. Michiels, P. Gorissen, and B. Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In C. Adams, A. Miri, and M. Wiener, editors, *Selected Areas in Cryptography*, number 4876 in Lecture Notes in Computer Science, pages 264–277. Springer Berlin Heidelberg, Jan. 2007.

[32] Y. Xiao and X. Lai. A secure implementation of white-box AES. In *2nd International Conference on Computer Science and its Applications, 2009. CSA '09*, pages 1–6, 2009.

[33] S. Yachi and M. Loreau. Biodiversity and ecosystem productivity in a fluctuating environment: the insurance hypothesis. *Proceedings of the National Academy of Sciences*, 96(4):1463–1468, 1999.