SCTE•ISBE CABLE-TEC EXPO® 2020
VIRTUAL EXPERIENCE » OCTOBER 12-15

2020 Fall Technical Forum
SCTE·ISBE • NCTA • CABLELABS®

Cloud-based Dynamic Executable Verification

A Technical Paper prepared for SCTE•ISBE by

Rafie Shamsaasef
CommScope
6450 Sequence Dr, San Diego CA 92121
858-404-2205
rafie.shamsaasef@commscope.com

Aaron Anderson
CommScope
117 St. Georges Bay Rd., Parnell
Auckland, New Zealand
+64 935 803 75
aaron.anderson@commscope.com

Sasha Medvinsky
CommScope
6450 Sequence Dr, San Diego CA 92121
858-404-2367
sasha.medvinsky@commscope.com

Table of Contents

Title **Page Number**

List of Figures

Title **Page Number**

List of Tables

1. Introduction

Modern software applications are composed of several inner connected modules enabling various features. Today's complex business and market-driven environment constantly pushes the edge to deliver software application faster than ever. Developers are battling with delivery deadlines that are not driven by the complexity of software offerings rather by the go-to-market motivations. As a result, insecure code has become a leading security risk and, increasingly, the leading business risk as well. It's irresponsible at every level to ignore this risk while doubling-down on anti-virus solutions and firewalls — neither of which protects applications [1].

It is important to have holistic view to software protection that provide check points and resolutions throughout the development cycle. It is also equally critical to empower the developers with technologies and methods to be able to automatically identify and detect certain types of attacks. There are commercial software security tools that transform cryptographic credentials so that they cannot be easily extracted. Other tools can make software reverse engineering very hard by sensing a debugger and transforming the binary code logic such that it looks unintelligible even with a debugger attached.

Dynamic Executable Verification (DEV) as described in this paper, provides low-impact dynamic integrity protection to applications that is compatible with standard code signing and verification methods. Further we discuss a system architecture where components of the Dynamic Executable Verification are placed into a secure cloud-based service which can only be configured by an authorized security administrator. To set the context, we discuss secure boot, tampering attacks and methods to perform static and dynamic analyses. Then we dive into details of DEV techniques that aim to ensure that software cannot be tampered with either statically or dynamically, without detection. The cloud aspect of the DEV makes it even easier for developers as the burden of configuring security tools is moved into a cloud service and the risk of releasing an application with lower than intended security is reduced. We will then present a couple of application use cases before concluding the paper.

2. Secure Boot Example and Limitations

Secure boot implies that each successive stage of software is authenticated – including BIOS or first-stage boot code, successive boot stage, Operating System kernel and all of the applications that execute on top of the OS. Authentication of each of the software layers is repeated every time that a system is re-initialized or rebooted. The very first set of instructions executed by a device after a reboot (first stage boot) may be protected in hardware as read-only (e.g., in ROM or a locked sector or flash) and may not need to be authenticated (since it cannot be changed).

Secure boot prevents physical tampering attacks that include opening up a device and re-flashing all of the software with unauthorized code that has not been digitally signed by an authorized party. If any piece of software is missing a digital signature or if its digital signature is invalid, the device will not boot – or at least that piece of software will fail to execute.

That's quite distinct from secure software download where a software image is authenticated once prior to being persistently installed in the device, into flash or a hard drive. Following a successful secure software download, that software is no longer validated even after a device reboots. A fully secured device would typically include both: secure software download for secure software updates and secure boot to prevent physical tampering with the device.

There are many different ways to achieve secure boot, but one example is illustrated in the figure below:

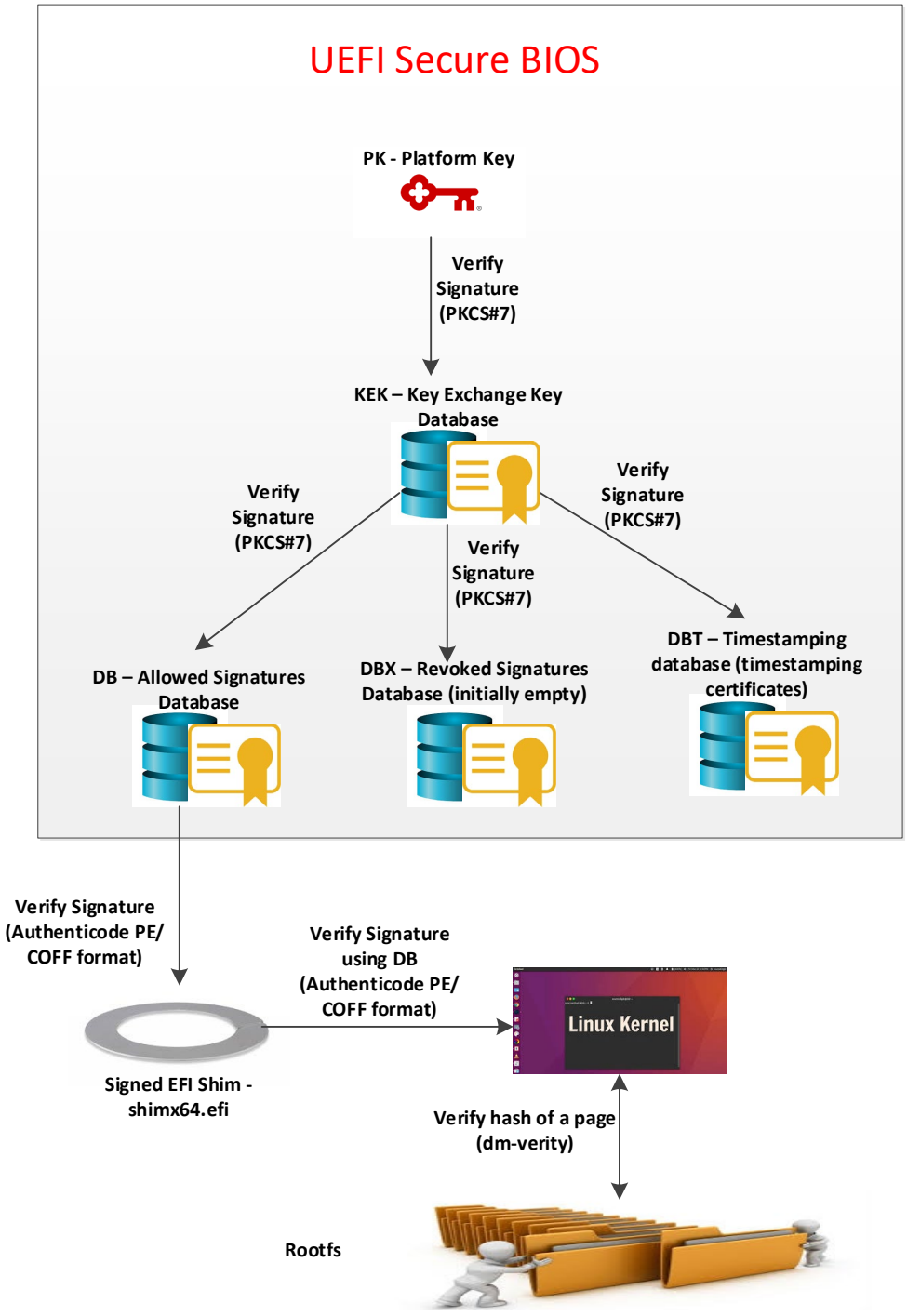

Figure 1 - Example of Linux Secure Boot

In this example, a standard UEFI BIOS that is now commonly available in Windows and Linux PCs as well as embedded devices has UEFI security turned on. It is assumed that UEFI BIOS is HW-protected and cannot be easily modified.[1] The figure shows the standard UEFI key/certificate hierarchy where the top-level Platform Key is used to verify the KEK (Key Exchange Key) which in turn is used to authenticate:

- DB (Allowed Signatures Database) that may for example contain a list of certificates that are permitted to validate a PKCS#7 signature.
- DBX (Revoked Signatures Database) may contain a list of code signatures that are on a prohibited list (e.g., due to security vulnerabilities in the corresponding code)
- DBT (Timestamping Database) – a list of certificates that may be used to validate signed timestamps, utilized to establish when a particular code release was signed. Timestamps may be utilized to reject an older version of the code with an earlier timestamp.

In this example, EFI Shim is first validated by a certificate inside DB and then launched and executed by the BIOS. EFI Shim in turn validates a signature on the Linux kernel (also using a certificate inside the DB) which is subsequently allowed to execute.

This is what is sometimes referred to as UEFI secure boot, but it is incomplete since none of the application code is authenticated and may be easily replaced by an adversary. One way to complete a Linux secure boot (as illustrated above) would be to:

- Place all applications, scripts and executables into a read-only file system such as a Linux rootfs.
- Configure the Linux kernel such that it contains a table of hashes of each page of the rootfs file system, using a facility called dm-verity [2]. Every time that a rootfs page is brought into RAM, its hash is validated. Any tampering to any of the code stored in rootfs will be eventually detected when the corresponding page is brought into RAM and it no longer matches the original hash of that page.

Such secure boot design can be achieved on some devices with custom firmware such as digital set-tops but in some cases, it is not possible to isolate all the executable code into a single read-only file system. Some devices by design have both executable code and data files stored within the same file system, some pages need to be dynamically changed and therefore page-level authentication techniques such as dm-verity do not apply.

In those cases, one cannot rely on secure boot to validate all of the binary code, especially application code, that may need to execute on a particular device. Techniques described in sections 6 and 7 of this paper provide alternative means to validate the more security-sensitive application code – even in the environments where full or even partial secure boot is not available.

3. Software Based Secure Boot Example

This next example of a software-based secure boot is simplified and has less software boot stage than you would normally see, but it illustrates how a software-based secure boot may be applied:

[1] Security guidelines for secure UEFI BIOS are provided in NIST SP 800-147

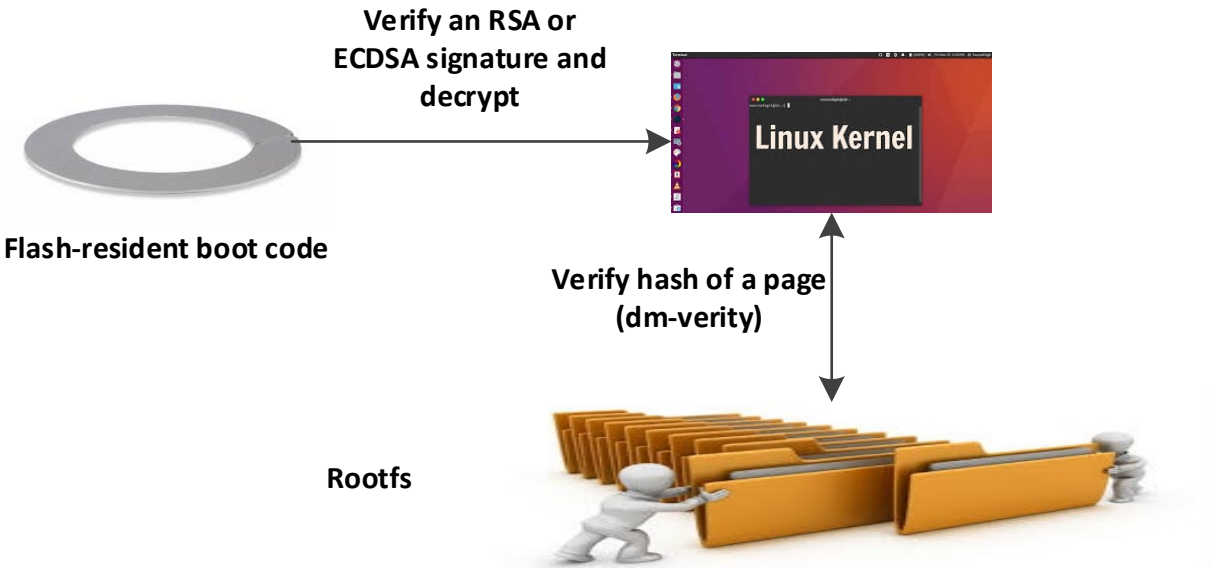

Figure 2 – Software Based Secure Boot Example

This device is relying on authenticity of the flash-resident boot code to verify integrity of the Linux kernel which in turn utilizes dm-verity to authenticate all the application code which is isolated in a read-only rootfs file system. This software boot architecture can provide secure boot only if we can be assured that the flash-resident boot code cannot be modified (e.g., by skipping a signature check on the Linux kernel).

Techniques presented in sections 6 and 7 of this paper may be utilized to prevent unauthorized modifications to the flash-resident boot code. Furthermore, this flash-resident boot code may be hiding a decryption key for the Linux Kernel using white box techniques to transform a cryptographic key. An attacker would have a hard time making changes to a self-validating version of the flash-resident boot code. However, replacing it completely would mean that the Linux kernel cannot be decrypted and extracting the decryption key for the Linux kernel is also made hard by white box techniques [3].

4. Static vs dynamic analyses

4.1. Static and dynamic analyses concept

Static and dynamic analyses of a software program are essential ways to discover software security vulnerabilities. They are both important elements of software security analyses and if done correctly, could expose cases that are more costly to resolve once the software is released to market. According to Wikipedia, “Static program analysis is the analysis of computer software that is performed without actually executing programs, in contrast with dynamic analysis, which is analysis performed on programs while they are executing.” [4]

Static analysis requires a good knowledge of software architecture and its internal structures of the software application. It is certainly the more thorough approach and may also prove more cost-efficient with the ability to detect bugs at an early phase of the software development life cycle. It goes deep into the software providing peace of mind that each and every line of source code has been thoroughly

inspected. It potentially can examine all possible execution path without running the program [5]. In a more sophisticated cases, static analysis can be performed on the binary version of the software revealing valuable information about all branches of the code.

Dynamic analysis, on the other hand, is capable of exposing a subtle flaw or vulnerability too complicated for static analysis alone to reveal [6]. It can generate real-time results of execution paths and provides valuable information to security analysts about the software behavior in the runtime environment. In the dynamic analysis, a set of canned input data are typically being supplied to analyze the behavior and expected outputs/results. If done in the controlled and configured settings, it can expose valuable security vulnerabilities in the runtime.

Static and dynamic analyses are only effective if they become part of the software development cycle and performed on every release. This requires certain security policies and practices to be in place per each software release iterations. Policies and practices are evaluated during the design phase from a security point of view. Then static analysis is performed by each developer per each development cycle to identify any leaking security holes. The process continues into the application execution phase where dynamic analysis is conducted in the runtime [3].

As part of this iterative process, developers and security engineers review the analysis report and come up with appropriate recommendations for the development team to consider. The process has to account for a systematic way to feed back the analyses results to the development team and make it a routine practice to adopt. The recommended changes will then be funneled thru the cycle as any other software features or bugs. This systematic and comprehensive process is essential part of any successful static and dynamic analyses practice.

4.2. Static and dynamic analysis tools

The quality and coverage of the analysis dependent on the sophistication of the analysis tools. It varies from those that only consider the behavior of individual source code statements and declarations, to those that include the complete source code of a program in their analysis. Tool can be deployed at unit level, system level or integration level. For a complete list of tools targeting various types of software applications refer to [7] and [8].

A comprehensive evaluation of analysis tool is beyond the scope of this paper. However, it is worth to mention the following criteria in choosing analysis tools:

Table 1 Static and dynamic analysis tools

Criteria	Description
Programming language	C/C++; Java; Perl; Python, etc.
Complexity of the code and inner logics	Code implementing mathematic, crypto and other complex algorithms
Input/output data media	Ways input and output data flow in and out of the application
Transport, network and communication mechanisms	TCP/IP, HTTP, etc.
Targeted Operating Systems	Linux, eLinux, Windows, Mac, etc.
Build environment	Host machine and cross compilers
Deployment platforms	Devices, servers, browsers, etc.
Virtual machines and containers	JVM, Docker, etc.
Obfuscation tools	Source code level or binary obfuscation
Configuration scripts	Settings and deployment scripts
Error handling and logging mechanisms	How errors and logs are exposed
Tool reporting capabilities	Reports and results of the analysis tool

5. Tampering Attacks and Threats

A consumer electronic device faces two types of general threats:

1) Outsider attacks performed by an attacker on the Internet or a public WiFi network. The attacker is for example able to scan the network for vulnerable devices containing outdated versions of the OS, web server, web browser or other commonly utilized applications which have not been recently patched.

   Once such a vulnerable device is found, at attacker is able to place her own attack software on that device. Attack software may be utilized to:
   - Compromise user's privacy – search user's system for passwords, credit card numbers and other private user information. Even if your own PC and electronic devices are well-protected, your private information may be stolen from one of many online banks, storefronts or other websites where your personal information is exposed. There are numerous and frequent examples of large-scale breaches, including Facebook [9], Yahoo [10], Google [11] and many-many more. And you are not always in control of protecting your personal electronic devices as security flaws may be exploited for a considerable period of time until the security flaw is discovered and patched by the manufacturer. [12]
   - Steal user's computing resources for attacker's own use such as Bitcoin mining (called Cryptojacking). User's device will appear to be very slow while much of its memory and CPU resources are utilized by this attacker [13].
   - Launch a distributed denial of service attack (after compromising many vulnerable devices) onto popular Internet services or storefronts – for political motives, a personal vendetta or just to create chaos on the Internet. Examples of such attacks that were carried out against GitHub, independent media sites in Hong Kong, CloudFlare security provider and content delivery network, Spamhaus anti-spam service and U.S. banks [14].

   Frequent patching of the OS and applications in the devices that you own and setting up perimeter security with firewalls and IP address filtering (including restricting ports and services) would provide a significant deterrent at least against direct attacks against you or your enterprise.

2) Insider attacks performed by a subscriber against electronic devices that are owned or leased by the subscriber – in order to rip off digital content, including making illegal copies of movies, songs and games after stripping off or disabling DRM protection.

   Additional insider attacks may be performed by disgruntled or dishonest employees, contractors or vendors. Such attacks are generally mitigated with authorization techniques – each person has restricted access only to the computing resources, applications and digital information that are required for that person to perform his or her job.

The focus of this paper is on advanced techniques for protecting software applications against tampering attacks during execution in runtime. Section 6 describes techniques to make software tamper-resistant independent of underlaying platform-related code signing protection and without requiring any HW security capabilities on the device where it executes. Section 7 goes further to protect key security parameters that are utilized in the generation of tamper-resistant code on a cloud server with restricted access. Developers are able to utilize a cloud service for creating self-verifying tamper-resistant code without direct access to security-sensitive parameters.

6. Dynamic executable verification design and concept

Any software application is potentially vulnerable to *tampering attacks* aimed at defeating their security measures [15] [16] [17]. Tampering attacks are a particularly low effort way to achieve license circumvention and software piracy. In particular buffer-overflow and hooking attacks are common dynamic tampering attacks that regular code-signing methods cannot detect or prevent. The last line of defense (and in many cases the only line of defense) is for applications to be capable of strongly defending their intellectual property and secrets against any such attacks. The goal of integrity protection is to increase the level of effort and complexity of such attacks.

We begin with a brief survey of integrity protection mechanisms in common use. We then describe a novel construction of dynamic executable verification.

6.1. Related work

In this section we briefly survey the methods of integrity protection in common usage:

6.1.1. Static code signing

The majority of integrity protection methods in commercial use are targeted at preventing *static tampering attacks,* which involve unauthorized modifications to a program's binary code prior to execution:

- Apple code signing [18].
- Microsoft code signing [19].

6.1.1.1. Drawbacks of static code signing

Code signing and verification methods do not detect *dynamic* modifications made to the executable code at runtime, such as with *buffer overrun attacks* [20].

6.1.2. Self-checking

Horne et al. [16] and Chang and Atallah [21] present self-checking techniques, in which a program repeatedly checks itself to verify that it has not been modified. These techniques consist of the dynamic (or runtime computation) of a cryptographic hash or a checksum of the instructions in an identified section of code, which is compared with a precomputed hash or checksum value at various points during program execution. Detected tampering will then trigger a response mechanism; such as a silent-failure-mode.

6.1.2.1. Drawbacks of self-checking

While such methods reliably detect unanticipated changes in the executable code at runtime, it is relatively easy for an attacker to identify the verification routine due to the atypical nature of the operation; since most applications do not read their own code sections [22].

Once detected, these schemes can usually be defeated with simple conditional logic modifications [21] or via hardware attacks [23]; and more recently by *virtual machine debugging attacks* [24], where the address ranges in a program's code section may be translated to an unchanged static image of the code so that any hash or checksum values are always computed correctly despite modifications to the underlying program code.

6.1.3. Just-in-time code decryption

Aucsmith [15] and Wang et al. [25] utilize the notion of self-modifying, self-decrypting code, where any tampering with the encrypted image will result in the decryption of "garbage" instructions, which leads to a catastrophic runtime failure.

6.1.3.1. Drawbacks of just-in-time code decryption

Several constructions using variations of this technique have been proposed and implemented [26] [27] [28] with varying results; however the widespread adoption of memory protection standards such as PAE/NX/SSE2 [29], and more recently, Intel's MPX [30] with support in mainstream operating systems and toolchains [31], limit this method to legacy and non-standard implementations. For example, since version 18.x of the Microsoft CL compiler/linker, the specification of a writeable attribute on executable code sections is ignored at both compile and in link time.

6.1.4. Oblivious hashing

Chen et al. [17] proposed a technique called *oblivious hashing*, where the idea is to hash an execution trace of a code section. The main goal is to blend the hashing code seamlessly with the code block, making it locally indistinguishable. An oblivious hash is *active* in the sense that the code to be protected must run (or be simulated) in order for the hash to be produced. An oblivious hash also depends on an exact path through a program, as determined by the program's inputs.

6.1.4.1. Drawbacks of oblivious hashing

Since an oblivious hash depends on a specific control-flow pathway in the executing program, this technique has limited applicability to specialist algorithms with simple linear control-flows. Additionally, since the computation of the oblivious hash is independent of other instructions being executed, tampering attacks that do not affect the internal control-flow of the program, such as *hooking attacks* [32], will remain undetected by this method.

6.1.5. Post-link executable modification

Many approaches involve the modification of executable code post-linking, so as to inject code or data elements used for the purposes of runtime verification, such as hashes or checksums.

6.1.5.1. Drawbacks of post-link executable modification

- Incompatibility with standard code-signing and verification methods.
- Limited toolchain compatibility due to possible conflicts with compile-time or link-time optimizations.
- Conflict with technologies that modify binaries post-linking, such as Apple's *application thinning* [33].
- Potential dependency on external third-party tools to finalize binary representations.

6.1.6. Other (intractable) approaches

Some methods, such as self-modifying code [34], appear in commercial usage without tractable security descriptions.

6.1.6.1. Drawbacks of intractable approaches

If the security properties of these methods cannot be objectively assessed, they are unlikely to pass a strict security audit.

6.2. Goals for integrity protection

We state the goals of integrity protection that we wish to satisfy with our construction.

1. Increase the cost of static and dynamic tampering attacks
    - Based on tractable principles that can be validated by a security auditor.
    - Resistant to real-word attacks.
    - Based on sound cryptographic principles.
2. Lightweight
    - Low performance overhead.
    - Small memory footprint.
3. Tunable
    - Fine-tunable percentage and locations of automatically generated integrity protection code.
    - Full manual control of integrity protection targets and the locations of verification code.
    - Automatic or custom detection responses.
4. Compatibility with standard code-signing

- Compatibility with standard code-signing and verification methods.

5. No frozen/finalized binaries
   - Frozen/finalized binary representations are not required.
6. Support a diverse range of use-cases
   - "Easy mode" where minimizing impact to existing processes is the primary motivation.
   - "Normal mode" for typical use-cases with reasonable expectations of security.
   - "Secure mode" for implementations where security is the primary motivation.
7. Future-proofed formats
   - Use of formats that provide future-proofing against changes to key sizes and cryptographic schemes, such as X.509.
8. Broad platform compatibility
9. Broad toolchain compatibility

6.3. Dynamic Executable Verification

In this section we describe a Dynamic Executable Verification (DEV) construction and its security properties.

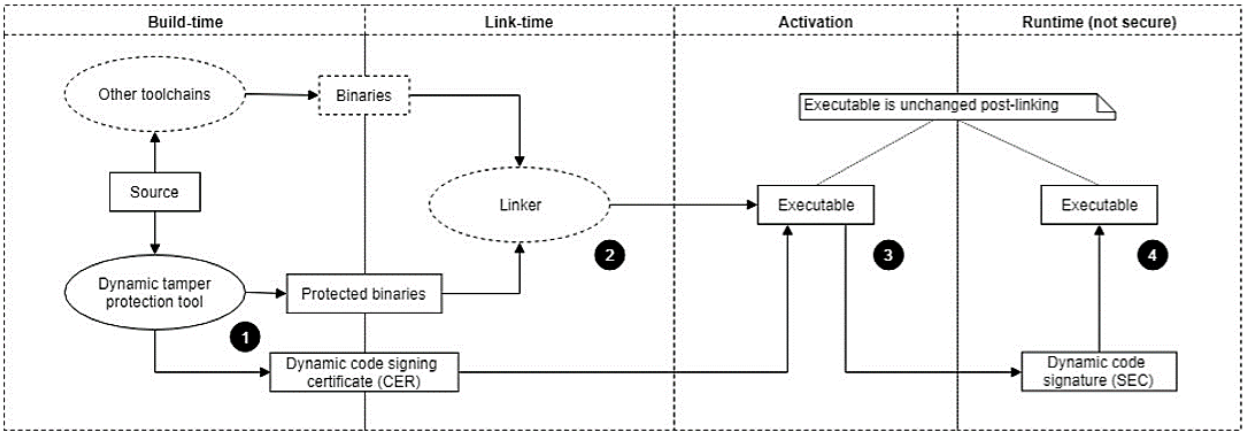

Figure 3 Dynamic executable verification generic use-case

1. Dynamic executable verification is added to an application at build-time to protect against tampering before and during runtime.
   - The tamper protection tool generates protected binaries along with a dynamic code signing certificate (CER).
   - The tamper detection code in the protected binaries is stealthy, diverse and spread throughout the application to mitigate discovery and circumvention of the tamper protection mechanics.
   - Unlike a static code signing certificate, the CER allows signing of dynamic code blocks within an executable to activate dynamic executable verification of that application.
2. Linking is done by a standard linker, which may incorporate unprotected binaries built by other toolchains.
3. Activation of dynamic executable verification is achieved by the executable using the CER to self-sign. This should be done in a secure environment to prevent the certificate from being leaked.

- If the certificate is valid and matches the protected portions of the executable, it generates a dynamic code signature (SEC) to be deployed along with the executable to the runtime environment.
- The executable is unchanged post-linking. This ensures maximum compatibility with sandboxed runtimes and traditional code-signing methods.

4. A runtime failure mode will be triggered if the executable or the SEC has been tampered with in any way before or during execution.

6.4. Construction

DEV protection is applied to a program (denoted by the symbol *P*) during compilation by the Dynamic tamper protection toolchain, resulting in a protected program (denoted by the symbol *P'*).

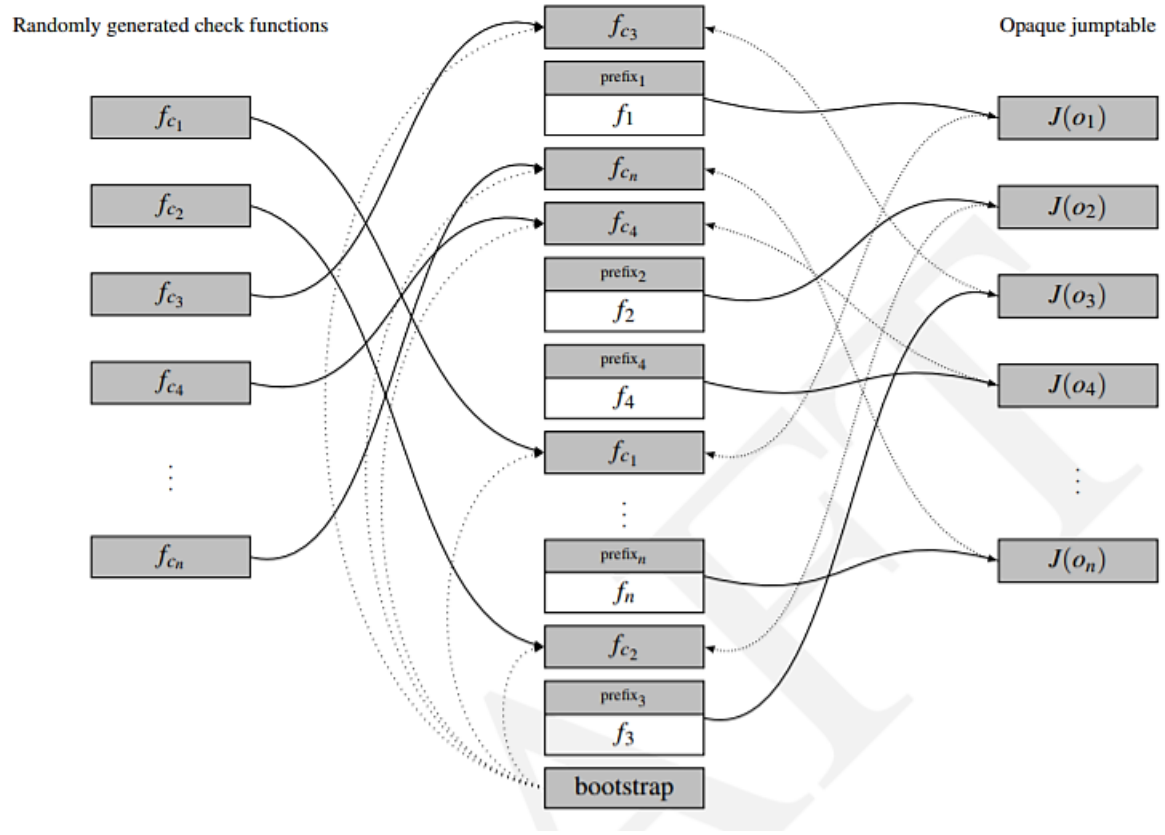

Figure 4 The DEV module injects random function prefixes (middle), check functions (left), an opaque jumptable (right), and a bootstrap (bottom) into the protected binary at build-time.

6.4.1. Random function prefixes

As depicted in Figure 4:

1. For each function/method *f* in the input program *P*, a random 16 byte function prefix f_{prefix} is prepended to *f* during compilation, using a standard LLVM operation [35]. We have verified that this method is compatible with all target platforms and toolchains; and with compiler and link-time optimizations.

2. LLVM ensures that the function prefix is aligned to the function's entry point, so that the injected checking code starts from the *prefix* address (typically flagged as an innocuous *data* section) rather than the actual address of the function *f*. This is designed to evade detection by automated and manual analysis techniques to identify self-referential tamper detection code [22].
3. At only 16 bytes per function, the use of a random function prefix is lightweight in terms of footprint, and has negligible impact on performance.

6.4.2. Randomly generated check functions

As depicted in Figure 4:

One or more check functions f_c (denoting the check function c of f) are randomly generated and injected at random locations in the protected binary P'

6.4.3. Opaque jumptable

As depicted in Figure 4:

1. The relationship between the calling function f, and check function f_c is obfuscated via the use of an opaque jumptable J.
2. For each randomly generated opaque identifier $o \in O$, the opaque jumptable computes the mapping $J(o) = f_c$ so as to conceal the relationship between the calling function f and the checking function.
3. Each checking function f_c references the prefix f_{prefix} of f rather than f itself, thus avoiding any direct reference to calling function f from the checking function f_c.
4. For added security, the mapping $J(o)$ may be implemented as a complex Boolean expression based on a reduction to a known hard problem, such as 3-CNF-SAT [36].

6.4.4. Bootstrap

As depicted in Figure 4:

1. If the signature (.sec) data does not exist, and if a valid X.509 certificate is passed to the bootstrap, then the bootstrap will *activate* DEV protection by calling each checking function $f_c \in P'$ (via the opaque jump table J) to generate the secure signature data.
2. Activation is carried out one-time per implementation instance.
3. Post activation, the bootstrap reads the .sec data into memory for use during runtime verification.

6.5. Runtime verification

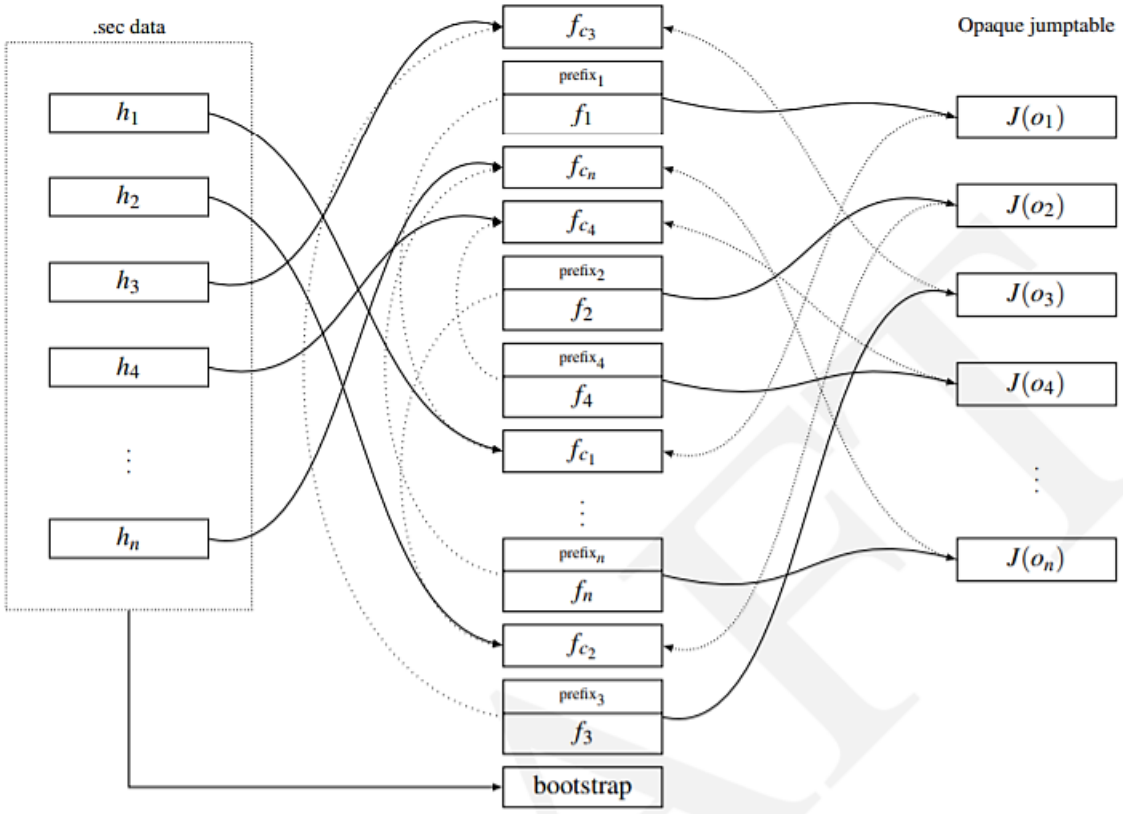

Figure 5 Dynamic executable verification happens during runtime execution, where each checker function is called according to the mapping defined in the opaque jumptable.

1. After DEV protection has been activated, the .sec data is used at runtime to enforce DEV integrity protection on all specified functions and methods.
2. Detected tampering with the executable or .sec data will result in a failure mode.
   - By default, the failure response should initiate a delayed system crash that is difficult for an attacker to track back to the detection code.
   - This failure mode may be overridden with a custom callback function by specifying the

6.6. Security

We have identified two primary security modes that can be utilized in conjunction with the activation method to achieve variable levels of security vs implementation overhead.

6.6.1. Mode 1

In mode 1, DEV protection is activated on the *first run* of the protected executable program. DEV protection does not modify this executable at any time. During activation, the DEV bootstrap validates the supplied DEV X.509 certificate (.cer) data. If valid, secure signature (.sec) data is generated, which is used by the executable to enforce integrity protection at runtime.

6.6.2. Mode 2

In mode 2, DEV protection is activated by a *setup executable*, which is separate from the runtime executable. DEV protection does not modify either executable at any time. During activation, the DEV bootstrap validates the supplied DEV X.509 certificate (.cer) data. If valid, secure signature (.sec) data is generated, which is used by the runtime executable to enforce integrity protection.

Security level	Mode	Activation	Runtime
Highest	2	Trusted	Untrusted
High	1	Trusted	Untrusted
Medium	1	Privileged	Untrusted
Low	1	Untrusted	Untrusted

Table 2 A range of security levels are attainable under different use-cases.

1. Activation in a trusted environment coupled with the splitting of activation vs runtime (in mode 2) provides the highest security level.
2. For high security applications in mode 1, DEV activation should be carried out in a trusted setting, so that the X.509 certificate remains confidential.
3. A medium security level is attainable with mode 1 if DEV activation is carried out in a factory setting or in a privileged runtime mode.
4. A low security profile is obtained in mode 1 if DEV protection is activated *on-the-fly* when the application is first executed at runtime.

7. Cloud-based architecture for dynamic executable verification

We present a cloud-based dynamic executable verification architecture to strengthen the overall security properties of DEV. In particular this will offer improved security of the dynamic code signing certificate (CER).

- The cloud service will only permit dynamic code signing request from authorized parties for authorized applications.
- The cloud service consists of two scenarios:
    1. Cloud virtual machine-based activation
    2. Local activation utilizing cloud-based code signing
- This service will be able to interact with other cloud-based security services to offer detailed reporting, metrics, and security alerts.
    - Detailed tamper protection coverage.
    - Activation failures.
    - Runtime metrics.
- Cloud-based dynamic code signing cannot be carried out by developers without cloud credentials or with insufficient permissions.

- A correctly defined permissions structure will ensure that only parties with the appropriate credentials can request dynamic signing for production deployment and that signing will only be permitted for applications built with valid developer credentials.
- There is no path to input manually generated CERs from non-cloud tamper protection tools into the cloud-based dynamic code signing. This is by design to prevent unauthorized developers from signing executables for production deployment.

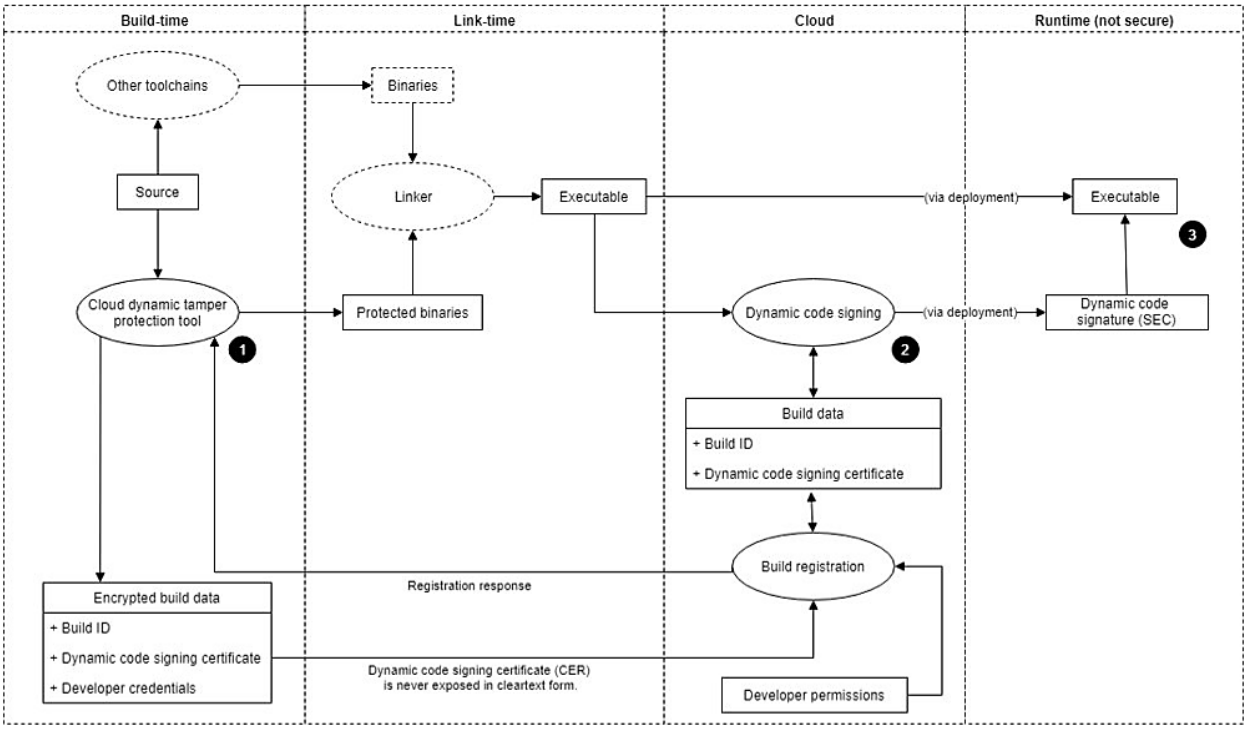

Figure 6 Cloud-based dynamic executable verification

1. At build-time, the cloud dynamic executable verification tool sends a unique Build ID and dynamic code signing certificate (CER) securely to the build registration cloud endpoint as depicted in Figure 6.
   - Cloud endpoint authenticates user's credentials before accepting the Build ID and CER over an encrypted link(e.g. using user name/password, digital certificate, one-time password, etc.).
   - Build registration will send a failure response if the request is not authentic or the developer credentials are not authorized for dynamic code signing.
   - Failure responses will abort the tamper protection tool with an error condition.
2. At activation time, the executable is securely sent to a dynamic code signing endpoint post-linking.
   - An authenticated connection to this endpoint is assumed to exist.
   - Cloud service runs executable in a secure VM to obtain a dynamic code signature (SEC).
   - If the executable is invalid, the dynamic code signing fails, or if the developer permissions associated with the Build ID are insufficient, the endpoint will abort with an error condition.
   - Otherwise, the endpoint returns the SEC for runtime deployment.

- Note that the dynamic code signing cloud endpoint does not accept a CER as input, thus preventing non-cloud protected executables from being signed.

3. A failure mode will be triggered if the executable or the SEC has been tampered with in any way before or during execution at runtime.
    - Note that the executable remains unchanged post-linking.

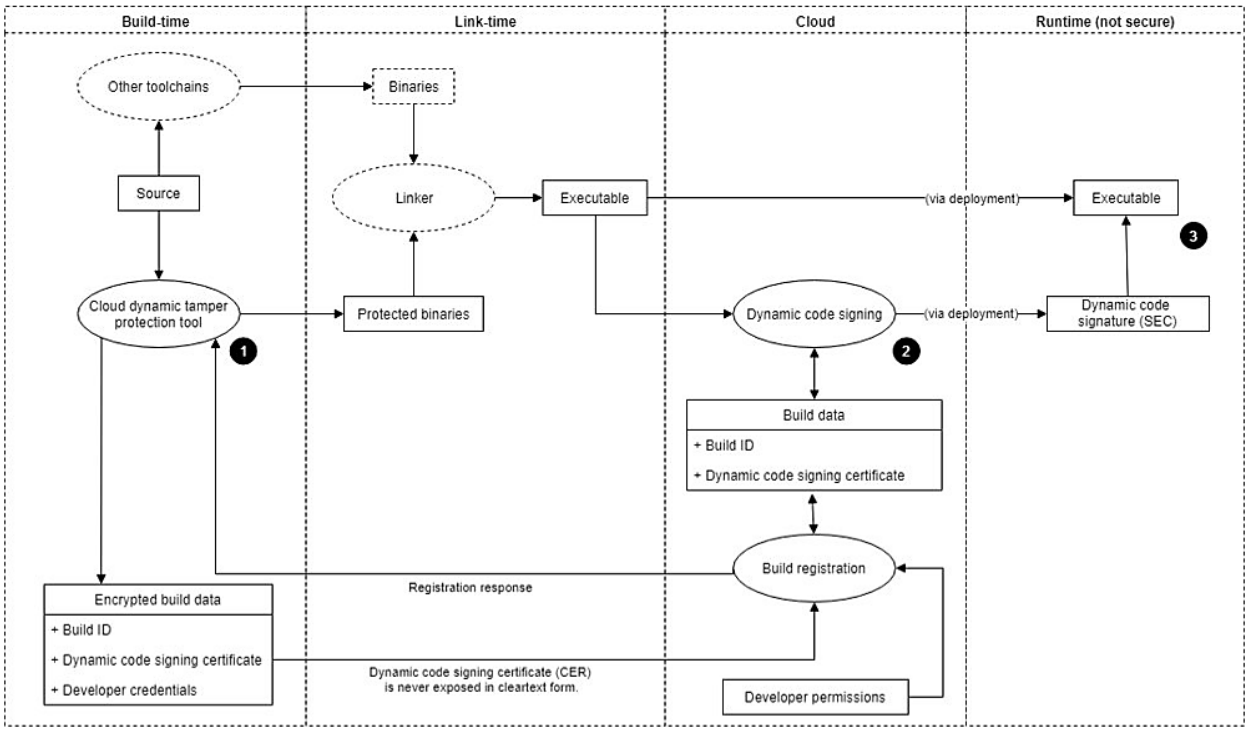

Figure 7 Cloud-based dynamic executable verification (no cloud VM)

1. In this scenario, the customer must set up a secure runtime environment to carry out the following.
    - Run a linked executable to generate a Dynamic Code Signing Request (DCSR).
    - Send the DCSR to the dynamic code signing cloud endpoint (via an authenticated connection).
2. The dynamic code signing cloud endpoint evaluates the DCSR.
    - An authenticated connection to this endpoint is assumed to exist.
    - If the request is not authentic, the DCSR is invalid or the developer permissions associated with the Build ID are insufficient, the endpoint will abort with an error condition.
    - Otherwise the endpoint returns a dynamic code signature (SEC) for runtime deployment.
    - As with the base scenario, the executable remains unchanged post-linking.
    - Note that the dynamic code signing cloud endpoint does not accept a CER as input, thus preventing non-cloud protected executables from being signed.

8. Application use cases

Dynamic executable verification discussed in this paper opens a great potential with a lot of oversights to designers/developers in protecting their applications. This technology provides ability to delivery software application with self-contained protection against tempering attacks without depending on the targeted platform security offerings. The ability to sign a portion of your software binary and dynamically verifying it in the runtime is applicable to any software application. A wide range of applications can utilize such techniques in their security software development practices and immediately obtain visibility into their application protection. Here we highlight a couple of these use cases.

8.1. Browser-based application

Browser-based application is a piece of software running in the browser context. It usually comes in a form of extension (i.e. Chrome) or plug-in (i.e. Edge) or even natively compiled binary (i.e. web assembly in Firefox). As a result, its live time and resource capabilities are limited to the browser session and prone to its attacks. Dynamic executable verification provides independent detection of browser session attacks to the application running in that context no matter which browser is hosting the application.

8.2. Container-based server application

As containers become more widespread and acceptable way of deploying server applications, more and more companies are migrating their application to utilize such environment. That of course comes with its own security risk and again dependency on the container provider. As a result, the application security relies on the security of underlaying container technology. Dynamic executable verification provides independent detection of attacks leaking out of hosting containers to applications. This provides a peace of mind to companies virtualizing their application to cloud.

8.3. DRM application

DRM agents or libraries hosted in an application can be subject for tampering attacks to lift and later user credentials and authorizations. Dynamic executable verification with cloud-based feature can actively detect and provide visibility to these types of attacks. It gives device/platform independent integrity protection and verification to strengthen the very core features of DRM applications.

9. Conclusion

In this paper, we discussed a software protection technology that can fill in the gap in securing applications without any dependencies on the targeted platform. Dynamic executable verification with cloud-based addition helps security engineers and developers to detect tempering threats throughout the code (as needed) in the runtime and take appropriate measures to remedy against them. This enables companies to deploy iterative security analysis processes with independent verifications in their software development practices, utilizing feedback received from attack surface analysis in runtime. There are all kinds of applications in various domains with different use cases that can benefit from this technology and take advantage of its features immediately. CommScope has deployed Dynamic executable verification technique in several products including DRM agents already and in the process of using cloud-based variation in near future.

Abbreviations

3-CNF-SAT	A Boolean satisfiability problem (SAT) in 3-Conjunctive normal form (CNF).
CER	Code signing certificate.
CSR	Code signing request.
DCSR	Dynamic code signing request.
DEV	Dynamic executable verification.
LLVM	Refers to the LLVM compiler infrastructure project.
SEC	Code signing signature.
UEFI	Unified Extensible Firmware Interface.

Bibliography and References

[1] Contrast Security, "A Modern Application Security Playbook," 2020. [Online]. Available: https://www.contrastsecurity.com/8-essential-steps-for-creating-security-strategy.

[2] M. Broz, "DMVerity," 2020. [Online]. Available: https://gitlab.com/cryptsetup/cryptsetup/-/wikis/DMVerity.

[3] A. A. Rafie Shamsaasef, "Dynamically Addressing the Gap of Software Application Protection without Hardware Security," in *SCTE Fall Technical Forum*, 2019.

[4] Wikipedia, "Static program analysis," [Online]. Available: https://en.wikipedia.org/wiki/Static_program_analysis.

[5] R. Kumar, "Dynamic code analysis VS Static code analysis," 2015. [Online]. Available: https://www.devopsschool.com/blog/difference-between-dynamic-code-analysis-and-static-code-analysis-2/.

[6] N. DuPaul, "Static Testing vs. Dynamic Testing," [Online]. Available: https://www.veracode.com/blog/2013/12/static-testing-vs-dynamic-testing.

[7] Wikipedia, "List of tools for Static Code Analysis," [Online]. Available: https://en.wikipedia.org/wiki/List_of_tools_for_static_code_analysis.

[8] S. Parker, "A list of dynamic analysis tools for software," 2019. [Online]. Available: https://www.peerlyst.com/posts/resource-a-list-of-dynamic-analysis-tools-for-software-susan-parker.

[9] A. Glaser, "Another 540 Million Facebook Users' Data Has Been Exposed," Slate, 2019. [Online]. Available: https://slate.com/technology/2019/04/facebook-data-breach-540-million-users-privacy.html.

[10] S. Larson, "Every single Yahoo account was hacked - 3 billion in all," CNN Business, [Online]. Available: https://money.cnn.com/2017/10/03/technology/business/yahoo-breach-3-billion-accounts/index.html.

[11] L. H. Newman, "A New Google+ Blunder Exposed Data From 52.5 Million Users," Wired, 2018. [Online]. Available: https://www.wired.com/story/google-plus-bug-52-million-users-data-exposed/.

[12] Forbes, "Google Shocks 1 Billion iPhone Users With Malicious Hack Warning," [Online]. Available: https://www.forbes.com/sites/zakdoffman/2019/08/30/google-shocks-1-billion-iphone-users-with-malicious-hack-warning.

[13] H. Tuttle, "Cryptojacking: How Hackers Steal Resources to Mine Digital Gold," Risk Management, 2018. [Online]. Available: http://www.rmmagazine.com/2018/08/01/cryptojacking/.

[14] A10, "5 Most Famous DDoS Attacks," [Online]. Available: https://www.a10networks.com/blog/5-most-famous-ddos-attacks/.

[15] D. Aucsmith, "Tamper Resistant Software: Design and Implementation," in *First International Workshop on Information Hiding*, 1996.

[16] B. Horne, L. Matheson, C. Sheehan and R. Tarjan, "Dynamic Self-Checking Techniques for Improved Tamper Resistance," *Security and Privacy in Digital Rights Management,* pp. 141-159, 2002.

[17] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha and M. Jakubowski, "Oblivious Hashing : A Stealthy Software Integrity Verification Primitive," *5th International Workshop on Information Hiding,* pp. 400-414, 2002.

[18] Apple Inc, "About Code Signing," 2012. [Online]. Available: https://developer.apple.com/library/mac/documentation/Security/Conceptual/CodeSigningGuide/Introduction/Introduction.html.

[19] Microsoft, "Introduction to Code Signing (Windows)," 2016. [Online]. Available: https://msdn.microsoft.com/en-us/library/ms537361.aspx.

[20] T. Schwarz and S. COEN, "Buffer Overflow Attack," 2004. [Online]. Available: http://www.cse.scu.edu/~tschwarz/coen152_05/Lectures/BufferOverflow.html.

[21] H. Chang and M. J. Atallah, "Protecting software code by guards," *Security and privacy in digital rights management,* pp. 160-175, 2002.

[22] M. Plasmans, "White-Box Cryptography for Digital Content Protection," 2005.

[23] P. C. V. Oorschot, A. Somayaji and G. Wurster, "Hardware-assisted circumvention of self-hashing software tamper resistance," *Distribution,* no. June, pp. 1-13, 2005.

[24] D. Quist and Valsmith, "Covert Debugging Circumventing Software Armoring Techniques," *Blackhat USA 2007 and Defcon 15,* 2007.

[25] P. Wang, S.-k. Kang and K. Kim, "Tamper Resistant Software Through Dynamic Integrity Checking," *Proc. Symp. on Cyptography and Information Security (SCIS 05),* 2005.

[26] J. Cappaert, N. Kisserli, D. Schellekens, B. Preneel and K. Arenberg, "Self-encrypting code to protect against analysis and tampering," in *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 2006.

[27] W. Thompson, "Cryptomorphic programming: a random program concept," *Florida State University, CS Dept., Advanced Cryptography,* vol. 131, pp. 1-11, 2005.

[28] W. Thompson, A. Yasinsac and J. T. Mcdonald, "Cryptoprogramming : A Software Tamper Resistant Mechanism Using Runtime Pathway Mappings".

[29] Hewlett Packard, "Data Execution Prevention," 2005.

[30] R. Ramakesavan, D. Zimmerman and P. Singaravelu, "Intel ® Memory Protection Extensions (Intel ® MPX) Enabling Guide," no. April, 2015.

[31] Intel Corporation, "Intel MPX support in the GCC compiler - GCC Wiki," [Online]. Available: https://gcc.gnu.org/wiki/Intel MPX support in the GCC compiler.

[32] S. Vogl, J. Pfoh, T. Kittel and C. Eckert, "Persistent Data-only Malware: Function Hooks without Code," *Network and Distributed System Security Symposium,* no. February, pp. 23-26, 2014.

[33] Apple Inc, "App Thinning (iOS, tvOS, watchOS)," 2016. [Online]. Available: https://developer.apple.com/library/ios/documentation/IDEs/Conceptual/AppDistributionGuide/AppThinning/AppThinning.html.

[34] G. Tropeano, "Self Modifying Code," *CodeBreakers Magazine,* vol. 1, no. 2, 2006.

[35] U. o. I. a. Urbana-Champaign, "LLVM: llvm::Function Class Reference," 2016. [Online]. Available: http://llvm.org/docs/doxygen/html/classllvm_1_1Function.html.

[36] L. A. Anderson, "A survey of control-flow obfuscation methods used in N-Mesh 2," no. October, 2015.

[37] H. Xu, Y. Zhou and M. R. Lyu, "N-Version Obfuscation: Impeding Software Tampering Replication with Program Diversity," *arXiv,* vol. arXiv:1506, 2015.

[38] B. Wyseur, "White-Box Cryptography," no. May, pp. 1-9, 2008.

[39] G. Wurster, P. C. Van Oorschot and A. Somayaji, "A generic attack on checksumming-based software tamper resistance," *Proceedings - IEEE Symposium on Security and Privacy,* pp. 127-135, 2005.

[40] H. Wee, "On obfuscating point functions," *Proceedings of the thirty-seventh annual ACM symposium on Theory of computing,* pp. 523-532, 2005.

[41] C. Wang, J. Hill, J. Knight and J. Davidson, "Software tamper resistance: Obstructing static analysis of programs," *Transform,* pp. 1-18, 2000.

[42] P. C. van Oorschot, "Revisiting Software Protection," *6th Int. Information Security Conf. (ISC 2003),* vol. 2851, no. October, pp. 1-13, 2003.

[43] P. C. Van Oorschot, "Overview – Software Protection," no. October, pp. 0-10, 2003.

[44] T. Tamboli, "Metamorphic Code Generation from LLVM IR Bytecode," *Thesis,* p. 72, 2013.

[45] N. Runwal, R. M. Low and M. Stamp, "Opcode graph similarity and metamorphic detection," *Journal in Computer Virology,* vol. 8, no. 1-2, pp. 37-52, 2012.

[46] T. Ogiso, Y. Sakabe, M. Soshi and A. Miyaji, "Software obfuscation on a theoretical basis and its implementation," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences,* Vols. E86-A, no. 1, pp. 176-186, 2003.

[47] J. Nagra, B. Wyseur and T. Herlea, "Trust Model for Software And Hardware-based TR methods," *RE-TRUST Deliverable D,* vol. 2, 2007.

[48] G. Myles and C. S. Collberg, "Software watermarking via opaque predicates: Implementation, analysis, and attacks," *Electronic Commerce Research,* vol. 6, no. 2, pp. 155-171, 2006.

[49] W. Michiels and P. Gorissen, "Mechanism for software tamper resistance: an application of white-box cryptography," in *Proceedings of the 2007 ACM workshop on Digital Rights Management*, 2007.

[50] T. Kerins and K. Kursawe, "A cautionary note on weak implementations of block ciphers," in *1st Benelux Workshop on Information and System Security (WISSec 2006)*, 2006.

[51] P. Junod, J. Rinaldini, J. Wehrli and J. Michielin, "Obfuscator-LLVM - Software Protection for the Masses," *Proceedings of the {IEEE/ACM} 1st International Workshop on Software Protection, {SPRO'15}, Firenze, Italy, May 19th, 2015,* pp. 3-9, 2015.

[52] M. Jakubowski, C. Saw and R. Venkatesan, "Tamper-Tolerant Software: Modeling and Implementation," *International Workshop on Security: Advances in Information and Computer Security (IWSEC '09),* pp. 125-139, 2009.

[53] S. Drape, *Contents Definitions of Obfuscation Evaluating Obfuscation Tools,* 2009.

[54] *CS 276 Lecture 11_ One-Way Functions _ in theory.*

[55] D. Corners, The Rootkit Arsenal, 2009.

[56] Cloakware Corporation, "Software protection and anti-tamper solutions for hostile environments Cloakware Security Suite Solution Overview".

[57] T. Brekne, "Encrypted Computation," 2001.

[58] J. M. Borello and L. Mé, "Code obfuscation techniques for metamorphic viruses," *Journal in Computer Virology,* vol. 4, no. 3, pp. 211-220, 2008.

[59] D. Baysa, R. M. Low and M. Stamp, "Structural entropy and metamorphic malware," *Journal in Computer Virology,* vol. 9, no. 4, pp. 179-192, 2013.

[60] L. A. Anderson, "Dynamic Executable Verification (DEV)," 2016.

[61] University of Illinois at Urbana-Champaign, "The LLVM Compiler Infrastructure. Copyright (c) 2003-2014. All rights reserved".

[62] University of Illinois at Urbana-Champaign, "Exception Handling in LLVM 3.8," 2016. [Online]. Available: http://llvm.org/releases/3.8.0/docs/ExceptionHandling.html.

[63] University of Illinois at Urbana-Champaign, "Cross-compilation using Clang," 2016. [Online]. Available: http://llvm.org/releases/3.8.0/tools/clang/docs/CrossCompilation.html.

[64] University of Illinois at Urbana-Champaign, "Clang command-line options," 2016. [Online]. Available: http://llvm.org/releases/3.8.0/tools/clang/docs/UsersManual.html.

[65] University of Illinois at Urbana-Champaign, "llc - LLVM static compiler," 2015. [Online]. Available: http://llvm.org/docs/CommandGuide/llc.html.

